
TeamCinder

Multi-Sensor Death Star Tracker

**Mechatronics 3 - 2007
Major Project**

Technical Manual



Date: 12 June 2007
Revision: 12 June 2007
Prepared by: Team Cinder (page2)
Distribution: MTRX 3700

Manufacturers' Details

Team Cinder



Bharath Mohan.....	305154044
Gerard Randika Dias.....	200410419
David Lamb.....	200411271
Wenjia Sun.....	305113070
Yannick Godin.....	305114921
Thomas Clement.....	200414799

CONTENTS

<u>Contents</u>	<u>Page No.</u>
TEAM CINDER.....	2
CONTENTS.....	3
1. INTRODUCTION	
1.1 – Document Identification.....	8
1.2 – System Overview.....	8
1.3 – Document Overview.....	8
1.4 – Reference Documents.....	8
1.5 – Acronyms and Abbreviations.....	9
2. SYSTEM DESCRIPTION	
2.1 – Introduction.....	10
2.2 – Operational Scenarios.....	10
2.2.1 – Failure Modes.....	11
2.3 – System Requirements.....	11
2.4 – Module Design.....	12
2.5 – Power Supply Module.....	13
2.5.1 – Functional Requirements.....	13
2.5.1.1 – Inputs.....	13
2.5.1.2 – Process.....	13
2.5.1.3 – Outputs.....	13
2.5.1.4 – Timing.....	13
2.5.1.5 – Failure modes.....	13
2.5.2 – Non-Functional (Quality of Service) Requirements.....	14
2.5.2.1 – Performance.....	14
2.5.2.2 – Interfaces.....	14
2.5.2.3 – Design Constraints.....	14
2.6 – Conceptual Design: Power Supply Module.....	14
2.6.1 – Failure modes.....	15
2.6.2 – Timing.....	15
2.6.3 – Outputs.....	15
2.6.4 – Process.....	15
2.6.5 – Inputs.....	15
2.7 – Functional Requirements.....	15
2.8 – LCD Module.....	16
2.8.1 – Non-Functional (Quality of Service) Requirements	
2.8.1.1 – Performance.....	16
2.8.1.2 – Interfaces.....	16
2.8.1.3 – Design Constraints.....	16
2.9 – Conceptual Design: LCD module....	16
2.10 – Ultra-Sonic Sensor Module.....	17
2.10.1 – Functional Requirements.....	17

2.10.1.1 – Inputs..	17
2.10.1.2 – Process.....	17
2.10.1.3 – Outputs.....	17
2.10.1.4 – Timing.....	17
2.10.1.5 – Failure modes.....	17
2.10.2 – Non-Functional (Quality of Service) Requirements.....	18
2.10.2.1 – Performance.....	18
2.10.2.2 – Interfaces.....	18
2.11 – Ultra Sonic Module.....	18
2.12 – Module Requirements: Infra-Red Module.....	19
2.12.1 – Functional Requirements.....	19
2.12.1.1 – Inputs.....	19
2.12.1.2 – Process.....	19
2.12.1.3 – Outputs.....	19
2.12.1.4 – Timing.....	19
2.12.1.5 – Failure modes.....	19
2.12.2 – Non-Functional (Quality of Service) Requirements	
2.12.2.1 – Interfaces.....	20
2.12.2.2 – Design Constraints.....	20
2.13 – Infra Red Module.....	20
2.14 – Module Requirements: User Interface Module (Serial).....	21
2.14.1 – Functional Requirements....	21
2.14.1.1 – Inputs.....	21
2.14.1.2 – Processes.....	21
2.14.1.3 – Outputs.....	21
2.14.1.4 – Timing.....	21
2.14.1.5 – Failure modes.....	22
2.14.2 – Non-Functional (Quality of Service) Requirements.....	22
2.14.2.1 – Interfaces.....	23
2.14.2.2 – Design Constraints.....	23
2.14.3 – Conceptual Design: Serial Module.....	23
2.14.3.1 – Flow diagram.....	23
2.14.3.2 – Testing and observation.....	24
2.14.3.3 – Buffer overflow.....	25
2.15 – Module Requirements: User Interface Module (Menu).....	26
2.15.1 – Functional Requirements....	26
2.15.1.1 – Inputs.....	26
2.15.1.2 – Processes.....	26
2.15.1.3 – Outputs.....	26
2.15.1.4 – Timing.....	26
2.15.1.5 – Failure modes.....	26
2.15.2 – Non-Functional (Quality of Service) Requirements	
2.15.2.1 – Performance.....	26
2.15.2.2 – Interfaces.....	26
2.15.2.3 – Design Constraints.....	27
2.15.3 – Conceptual Design: Menu Module.....	27
2.15.3.1 – Initial Menu Design – Menus.....	28
2.15.3.1 – Initial Menu Design	
2.15.3.1.1 – Menu Selection.....	31
2.15.3.1.2 – String Updating.....	31
2.15.3.1.3 – Initial Menu Design– Variable Strings	

2.15.3.1.4 - Storing Keypad Values.....	31
2.15.3.1.5 - Menu Transitions.....	31
2.15.3.1.5 - Logbook Entries.....	33
2.15.3.1.6 - Functional Approach.....	33
2.16 - Module Requirements: Keypad Module.....	33
2.16.1 - Functional Requirements.....	33
2.16.1.1 - Inputs.....	33
2.16.1.2 - Processes.....	33
2.16.1.3 - Outputs.....	33
2.16.1.4 - Timing.....	34
2.16.1.5 - Failure modes.....	34
2.16.2 - Non-Functional (Quality of Service) Requirements	
2.16.2.1 - Performance.....	34
2.16.2.2 - Interfaces.....	34
2.16.2.3 - Design Constraints.....	35
2.16.3 - Conceptual Design: Keypad Module.....	36
2.17 - Module Requirements: 3D Tracking and Scanning	
2.17.1 - Functional Requirements.....	38
2.17.1.1 - Inputs.....	38
2.17.1.2 - Process.....	38
2.17.1.3 - Outputs.....	38
2.17.1.4 - Timing.....	38
2.17.1.5 - Failure modes.....	38
2.17.2 - Non-Functional (Quality of Service) Requirements	
2.17.2.1 - Performance.....	39
2.17.2.2 - Interfaces.....	39
2.17.2.3 - Design Constraints.....	40
2.18 - Conceptual Design: 3D Tracking and Scanning Module.....	40
3. USER INTERFACE	
3.1 - Classes of User.....	42
3.2 - System Functionality.....	42
3.2.1 - User Local Mode.....	42
3.2.1.1 - Option 1: Scan.....	43
3.2.1.2 - Option 2: Menu.....	45
3.2.2 - User Remote Mode.....	47
3.2.3 - Factory Mode	
3.2.3.1 - Entering factory Mode from User Local Mode.....	49
3.2.3.2 - Entering factory Mode from User Remote Mode.....	50
3.3 - Shut-down Sequence.....	52
3.4 - Interface Design: User Class Y.....	52
3.4.1 - User Inputs and Outputs.....	52
3.4.2 - Input Validation and Error Trapping.....	53

4. HARDWARE DESIGN

4.1 - Scope of the DST System Hardware.....	54
4.2 - Hardware Design.....	54
4.2.1 - Power Supply.....	54
4.2.1.1 - Circuit Schematic....	55
4.2.2 - Internal Connections (Pin Allocations)	56
4.2.3 - Sensor Hardware.....	57
4.2.3.1 - US Sensor.....	57
4.2.3.2 - Sharp Infrared Sensor Infra-Red Sensor.....	57
4.2.3.3 - National Semiconductor IC Temperature Sensor.....	58
4.2.4 - Actuator Hardware.....	59
4.2.4.1 - Servos.....	59
4.2.5 - Operator Input Hardware.....	60
4.2.5.1 - Keypad.....	60
4.2.6 - Operator Output Hardware.....	61
4.2.6.1 - LED Scanning & Tracking.....	61
4.2.6.2 - LCD.....	62
4.2.7 - Hardware Quality Assurance.....	64
4.3 - Hardware Validation.....	64
4.4 - Hardware Calibration Procedures.....	64
4.5 - Hardware Maintenance and Adjustment.....	64

5. SOFTWARE DESIGN

5.1 - Software Design Process.....	65
5.2 - Software Development Environment.....	65
5.3 - Software Implementation Stages and Test Plans.....	65
5.4 - Software Quality Assurance.....	67
5.5 - Architecture.....	68
5.5.1 - State transition diagram.....	68
5.5.2 - PWM module.	69
5.5.3 - Interrupt Handlers.....	69
5.5.4 - Tracking and scanning transition.....	71
5.6 - Software Interface.....	72
5.7 - Software Components.....	72
5.8 - Preconditions for System Start-up.....	72
5.9 - Preconditions for System Shutdown.....	72

6. SYSTEM PERFORMANCE

6.1 - Performance Testing.....	73
6.1.1 - IR and US Sensor.....	73
6.1.2 - Temperature sensor.....	73
6.1.3 - Memory Usage.....	73
6.2 - State of the System as Delivered.....	73
6.3 - Future Improvements.....	74

7. SAFETY IMPLICATIONS

7.1 – Operational Conditions.....	75
7.2. User's Perspective.....	75
8. CONCLUSIONS.....	76
APPENDIX A:	77
APPENDIX B:	78
APPENDIX C:	80
APPENDIX D:	84
APPENDIX E:	87
APPENDIX F:	88
APPENDIX G: Product Cost Listing.....	89
DOXYGEN: Function Listings.....	90

1. INTRODUCTION

1.1 - Document Identification

This document describes the design of a Multi-sensor Death Star tracker. This document is prepared by Team Cinder for assessment in MTRX 3700 in 2007.

1.2 - System Overview

This document describes the operation of the Death Star Tracker – a system which can scan, detect and track an object in a limited range of free space to output a distance measurement reading. The system is calibrated to function under temperatures ranging from 0-40 degrees Celsius.

1.3 - Document Overview

This document is provided to give the user an in depth understanding on the working of this system; i.e.:

- ❖ Program structure
- ❖ Specifics on hardware and software interfacing
- ❖ How tracking and scanning are implemented
- ❖ How an accurate distance measurement can be achieved.

Upon careful analysis of this document, the user will be able to understand how each step of the program functions and how systems such as this are designed in the real world.

For a list of contents refer to page 3

1.4 - Reference Documents

The present document is prepared on the basis of the following reference documents, and should be read in conjunction with them. All The following documents are available at <http://www.acfr.usyd.edu.au/teaching/3rd-year/mtrx3700-Mx3/labs/index.html>.

- ❖ PIC18F452 Datasheet (5.881 MB PDF File)
- ❖ Polaroid 6500 Series Sonar Ranging Module (253 KB PDF File)
- ❖ TI TL851 Sonar Ranging Control (83 KB PDF File)
- ❖ TI TL852 Sonar Ranging Receiver (95 KB PDF File)
- ❖ Polaroid 600 Series Instrument Transducer (226 KB PDF File)
- ❖ Sharp GP2Y0A02 Datasheet (65 KB PDF File)
- ❖ Notes on RC Servos (45 KB PDF File)
- ❖ Information on Alphanumeric LCD Modules (includes data sheets)
- ❖ LM35DZ Datasheet (201 KB PDF File)

1.5 - Acronyms and Abbreviations

Acronym	Meaning
ACFR	Australian Centre for Field Robotics
CTS	Continuous <i>or</i> Continuously
US	Ultra Sonic
TBD	To Be Done, <i>also</i> To Be Defined
IR	Infra-Red
USYD	The University of Sydney
DST	Death Star Tracker
LCD	Liquid Crystal Display

2. SYSTEM DESCRIPTION

This section is intended to give a general overview of the basis for the Death Star Tracker system design, of its division into hardware and software modules, and of its development and implementation.

2.1 – Introduction

Several modules work together to make the DST operational. All the main system logic is conducted on the PIC18F452 which is mounted on the MNML PIC 18 circuit board, located in the sensor array enclosure.

An object (death star) is moved manually in-front of the systems sensor array to trigger the tracking mode which allows the user to obtain an accurate position and distance measurement from a pre-calibrated zero position in free space.

The User I/O software module controls how the system behaves when faced with various input possibilities applied by the user.
The input is provided either via buttons on the User Interface Enclosure or the serial connection.

The serial module consists of a custom designed communication protocol allowing the user to perform the same input operations using both the User Control Interface (User Local mode) and the serial connection (User Remote and Factory Mode). The serial module also allows the user a more advanced interaction with the system (Factory Mode) for further elemental calibration.

User output is via a number of sources; LED's, LCD, Buzzer and HyperTerminal

2.2 - Operational Scenarios

There are two broad classes of users: Consumers and the Technical user.

The average consumer will be able to access only the basic level of functionality that this product offers, i.e.: only User Local and User Remote mode will be available.

Technical users can interact using all the methods plus the additional Factory mode which can affect the performance of the system itself.
Technical users will be the technicians involved in factory calibration or maintenance. They will generally not use the system for the product that it is, but will rather correct and test the operation of the system.

2.2.1 - Failure Modes

The following failures may occur. The consequences of each failure are also briefly outlined.

- ❖ Failure of the buttons on the Sensor Array interface will stop the system turning on and/or charging the battery for use.
- ❖ Failure of the buttons on the User Interface Enclosure will result in the entire system being made redundant, as upon startup, the system will enter User Local mode (4x3 keypad data input) and can only be changed using the buttons. The buttons are required to activate the serial communication.
- ❖ Failure of the LCD screen will result in all user Local mode output non-existent. Only a technical user will be then able to switch modes and operate the system without the LCD output.
- ❖ Total software failure, will mean no system functionality and the product will be useless
- ❖ Failure of product shutdown will result in component overheating and ultimately lead to product failure.

2.3 - System Requirements

The operational scenarios considered place certain requirements on the whole Multi-sensor Death Star Tracker system and on the modules that comprise it.

First, and foremost, the system requires a sufficient, reliable power supply to all modules.

The most important module is the user interface module; broken down into Serial Interface and the buttons on the User Interface enclosure (Keypad and rocker switch). Corruptions to this module in either software or hardware will cause the whole scale system to operate in an unpredictable manner.

The following requirements are therefore essential for the system to work:

- ❖ Functional hardware and software
- ❖ Reliable connection between the system and the PIC board.

The ability of the sensor array to operate properly comes next in system importance. If the sensors are not working, it will directly affect the functionality of the user interface module and the product will once again be made useless.

Other hardware and software modules are considered less important to the state of the system as a whole since basic operation can still be achieved. Although a failed

output module will not cause the system to malfunction, it will restrict the systems capabilities, thus reducing the users' interaction.

2.4 - Module Design

The modules for this system were split-up as shown below: (in no specific order)

- ❖ Power Supply Module
- ❖ LCD Module
- ❖ Ultra-Sonic Sensor Module
- ❖ Infra Red Sensor Module
- ❖ Serial Module
- ❖ Keypad Module
- ❖ 3D Scanning and Tracking Module

2.5 - Module Requirements: Power Supply Module

The power supply hardware module creates a +5V source to power the system. It has 'hardware off, software on' control capability (PIC MNML board always running).

2.5.1 - Functional Requirements

This section describes the functional requirements of the Power Supply module – those requirements that must be met if the module (and system) is to function correctly.

2.5.1.1 – Inputs

The power supply module is located within the Sensor Array Enclosure and provides a 12V power source to the entire system. It is a 12V, 4.2AH (amp hour) SLA battery which is then regulated to 5V for product use. The system can be re-charged using an ordinary 240V power socket with the 12V SLA charger (NB: Centre tip Negative).

2.5.1.2 – Process

The charger provides the system with sufficient power to run all the components whilst having the device remote to a power supply. This system allows for over 12hours continuous running time. Once the system begins to reduce in charge, the user can then plug in the 12V charger and run the device off the mains power whilst at the same time charging the battery. The battery can also be charged stand alone (system off recharge on).

2.5.1.3 – Outputs

The output from this module is a regulated 5V supply. This output is generated to run all the components off the device at a stable voltage using a 7805 voltage regulator. Having the ability to handle 1amp current capacity, the 7805 enables the device to run efficiently as long as a by pass capacitor is present for the running of the ultra sonic sensor.

2.5.1.4 – Timing

Constant output of a regulated 5V when this module is functioning.

2.5.1.5 - Failure modes

Power module failure can only be fixed with replacement. Failure would mean the redundancy of the entire system

2.5.2 - Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

2.5.2.1 – Performance

The SLA battery functioned well on all occasions.

2.5.2.2 – Interfaces

Output current should be kept to minimum to reduce heating problems. The power from the battery was regulated to 5V before dispensing to the rest of the components. A Power grid was also set up for ease of use.

2.5.2.3 - Design Constraints

The battery was purchased at a reasonable cost.
All wires connecting the module to the rest of the system are readily available to all users.
Power must be regulated before input to other hardware elements.

2.6 - Conceptual Design: Power Supply Module

The power source had to be wired in such a way that it would charge and power the system at the same time. We utilized two rocker switches to toggle each of the functions on and off when required. Both power out and power in to the battery had to work in unison.

2.7 - Module Requirements: LCD Module

The operational scenarios considered place certain requirements on the LCD Module and on the subs systems that comprise it.

2.7.1 - Functional Requirements

This section describes the functional requirements of the LCD module – those requirements that must be met if the module (and system) is to function correctly.

2.7.1.1 – Inputs

The LCD module is located on the User Interface Enclosure and takes power directly from the regulated 12V source (SLA battery). Its operation is ultimately dependant on the right amount of power being applied to the system to display characters and switch on the backlight.

The software inputs to the LCD module are the two strings determined from the menu module to be sent to the LCD

2.7.1.2 – Process

Sends array to the specified string function and takes each character and inputs it one by one with a delay.

At a software level the LCD triggers a string update if and only if the new strings are different from the old ones. This is taken care of in the menu module, and so the only process undergone in the LCD module is the sending of strings to the LCD

2.7.1.3 – Outputs

The output of the LCD module is an illuminated light LCD with 2 lines of 16 characters each.

2.7.1.4 – Timing

The power to the LCD must first be initialised before software can be sent to it for constant updates. When the Red power switch is toggled, all hardware components including the LCD are powered up before software is called on.

2.7.1.5 - Failure modes

As mentioned in 2.2.1, if LCD fails to initialise then the user will have no output in User Local Mode except for the buzzer sounding every time the keypad triggers (if enabled).

2.7.2 - Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

2.7.2.1 – Performance

There are no strict performance requirements; only that the LCD should be bright enough to be easily read i.e. supplied with the correct power.

2.7.2.2 – Interfaces

The wires to and from the LCD system are well housed and protected from external interference. Although they are not an integral part of the required functionality, faults in their existence could cause the system to malfunction.

2.7.2.3 - Design Constraints

The LCD was purchased at a reasonable cost and all wires connecting the system to the rest of the product are readily available to all users. Any configuration of characters can be displayed to suite users needs upon manufacturing.

2.8 - Conceptual Design: LCD module

The LCD supports both 4-bit and 8-bit data busses. A 4-bit data bus (shown in section 2.43 - pin allocation) was nominated for use to reduce the number of pins used for this module. These free pins allowed extra functionality to be incorporated into the product. Data is transmitted in two nibbles. The most significant four bits are transmitted and then the least significant four bits of a byte are transmitted to communicate.

Detail on this communication protocol can be found in the manufacture's data sheet.

This module is totally self contained and does not rely on global variables or other modules. Such design allows this module to be integrated and used in other code applications.

The code was generated using the Application Maestro (under the microchip menu) function of MPLAB by setting the correct parameters and generating code. With this module, it was easy to incorporate the menu update system where a single function, 'XLCDPutRamString', can be called to send strings from data memory to the LCD screen. Each line is defined by the functions; 'XLCDL1home();' and 'XLCDL2home();'

2.9 - Module Requirements: Ultra-Sonic Sensor Module

The operational scenarios considered place certain requirements on the Ultra-Sonic Sensor Module and on the sub systems that comprise it.

2.9.1 - Functional Requirements

This section describes the functional requirements of the Ultra Sonic module – those requirements that must be met if the module (and system) is to function correctly.

2.9.1.1 – Inputs

The 6500 Series module operates over a supply range of 4.5 volts to 6.8 volts and is characterized for operation from 0° C to 40° C

2.9.1.2 – Process

The 6500 Series is an economical sonar ranging module that can drive all SensComp/Polaroid electrostatic transducers with no additional interface. This module, with a simple interface, is able to measure distances from 6 inches to 35 feet. The typical absolute accuracy is (+-) 1% of the reading over the entire range.

Driving INIT pin high will send out a sound wave, ECHO pin will be driven to high if a return sound wave has been detected. Time elapsed to receive an echo back will be captured by hardware using CCP module to provide accurate distance reading.

2.9.1.3 – Outputs

This module outputs a value of TMR3 (16bits) which is captured by CCP module after an echo wave has been received. It is this reading which is used to calculate object distance.

2.9.1.4 – Timing

The module has an accurate ceramic-resonator-controlled 420-kHz time-base generator. An output based on the 420-kilohertz time base is provided for external use. The sonar transmit output is 16 cycles at a frequency of 49.4 kilohertz.

2.9.1.5 - Failure modes

High Sampling Rate might cause the entire sensor module fail which will negatively affect the entire system.

If the US sensor fails, the tracking feature of the product will become redundant.

As for distance readings, the range will be limited to IR's capabilities and the values will decrease in accuracy.

2.9.2 - Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

2.9.2.1 – Performance

The US sensor relies on the correct amount of power being inputted to the system for optimal performance. Tracking and distance measurement calculations can only be accurately executed if the US sensor is working to specifications.

2.9.2.2 - Interfaces

The US sensor is connected to a 5V regulator circuit to filter the raw 12V power supplied by the SLA battery.

2.9.2.3 - Design Constraints

The US had to be run of specific pins to account for its requirements. Refer to section 4.2.2 for pin allocations.

2.10 - Conceptual Design: Ultra Sonic Module

Operation of the Ultra Sonic Module relies upon the external edge triggered interrupt feature available using the PIC microcontroller, and that of the CCP2 timer. The sample rate is determined by a flag and counter in the overflow triggered interrupt routine that fires every 10ms. This flag is labelled `US_sample_counter`, and the counter is labelled `no_overflows`. `No_overflows` is incremented every overflow interrupt (every 10ms), and when it reaches the value defined by `US_sample_counter`, a flag that allows a pulse to be fired is set. This way, the sample rate can be increased or decreased by increments of 10ms per pulse.

In the main program loop, there is a check of the flag that allows a pulse to be fired (`send_pulse_flag`). When this flag has been set high during the overflow interrupt, 5V is sent to the INIT pin that fires a pulse, and `send_pulse_flag` is reset to zero. Immediately after this, TMR3 is started. The ECHO pin from the module is connected to an edge triggered interrupt, and this will fire when an echo is received. Within this interrupt, TMR3 is immediately stopped, and flag that allows distance calculation is set (`calculate_flag`). Back in the main loop, there is a check of this `calculate_flag`, and if it is set then there is a mathematical operation to convert the number in TMR3 to a distance value. Once this is complete, TMR3 is and `calculate_flag` are reset to zero. The module then stays idle until enough overflow interrupts fire to begin the process again.

2.10 - Module Requirements: Infra-Red Module

The operational scenarios considered place certain requirements on the Infra-Red module and on the sub systems that comprise it.

2.10.1 - Functional Requirements

This section describes the functional requirements of the Infra Red module – those requirements that must be met if the module (and system) is to function correctly.

2.10.1.1 – Inputs

The Sharp GP2Y0A02 Infra Red sensor is operated by a Supply Voltage of 4.5→ 5.5V. The voltage is drawn from a 12V SLA battery pack which is then regulated to output and constant 5V

2.10.1.2 – Process

The Infra-Red module automatically sends out Infra-red signals when powered, and the IR sensor on the module will measure the strength of the returning IR beam and convert this into a voltage value.

2.10.1.3 – Outputs

This module will send the 'return' voltage value from the IR sensor to the A/D converter, where a digital value between 0x0000 and 0xFFFF will be received. This value forms the integral part of a mathematical equation to determine the strength of the returning beam, thus finding the distance that the Death Star is from the sensor array.

2.10.1.4 – Timing

The IR hardware is continuously sending IR pulses and receiving IR signals. The timing of the module depends on how often the AD/converter is assigned to read a value returning from the IR sensor. The frequency of this sampling is determined by a flag titled 'IR_overflows', which, when it reaches a number titled 'IR_sample_counter' triggers a flag to begin an A/D conversion. IR_overflows is incremented every time the overflow interrupt is triggered (every 10ms), and so setting the value for 'IR_sample_counter' can set the timing/frequency of taking IR measurements in degrees of 10ms.

2.10.1.5 - Failure modes

IR 'failures' are detected by checking the value that is returned from the A/D conversion. If less than 0.5 volts are detected as a measurement, then this output value is disregarded as the target is most likely out of range of the module, and the distance variable is set to zero.

2.10.2 - Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

2.10.2.1 – Performance

The IR module has no specific performance requirements except to generate a voltage from the sensor. If there is insufficient power supplied to the module, then its operation is undefined, and therefore there is not exception for performance requirements

2.10.2.2 – Interfaces

The only interface of the IR module with the rest of the system comes via the 3 pins for power (V_{cc}), ground (Gnd), and the output (V_{out}). If all these 3 are not connected properly, then the system will not operate correctly, so there is no interface requirement that non-functional.

2.10.2.3 - Design Constraints

The IR sensor module had to be run off specific pins to account for its requirements.

Refer to section 4.2.2 for pin allocations.

2.11 - Conceptual Design: Infra Red Module

The module will exist in hardware without any alterations to the supplied equipment. Details on this hardware can be found in the manufacture's data sheet. In hardware, the IR module will be set up by sending 5V to the module from an external regulator, 0V at the ground pin by linking it to the common ground node of the system in hardware, and the output sent to the microcontroller board via pin RA1 on PORTA.

In software, the pin PA1 is set to input by setting TRISA1 to 1, and the A/D module is configured to enable input from RA1, and enable conversion complete interrupt triggering. Within the low priority interrupt routine, there will be a check to see if the interrupt has been triggered by an overflow, and will increment an IR counter if so. When this reaches the value determined by `IR_sample_counter`, a 'begin A/D conversion' flag will be set. This method enables changing the IR measurement sample rate by changing the value of `IR_sample_counter`.

In the main program loop, if begin conversion set, then the A/D converter GO bit is set, triggering an A/D conversion. Once this is complete, the low priority interrupt will trigger again, and a 'IR_calculate' flag will be set. Again in main code, if this bit is set the program will convert the A/D result into a value for distance of the Death-Star from the IR module, updating the 'IR_distance' value using the equations shown in Appendix A.

However, if the voltage read received by the IR module was less than 0.5V, then the `IR_distance` is set to zero.

2.12 - Module Requirements: User Interface Module (Serial)

The operational scenarios considered place certain requirements on the Serial Module and on the sub systems that comprise it.

2.12.1 - Functional Requirements

This section describes the functional requirements of the Serial Module – those requirements that must be met if the module (and system) is to function correctly.

2.12.1.1 – Inputs

The Serial module has only one form of input. The device is connected to a PC computer by means of a 9 pins serial cable and inputs can be entered through the keyboard by opening a HyperTerminal window at 9600 BAUD.

2.12.1.2 – Processes

The serial communication module is entirely interrupt driven and uses transmit and receive functions to process the user input. It generates the appropriate response by controlling the behavior of the device and displaying feedback on the HyperTerminal window.

2.12.1.3 – Outputs

- ❖ Strings displayed on the HyperTerminal window.
- ❖ Should the HyperTerminal window become full, the user has the option to refresh the page by pressing the key “r” in both user remote and factory mode.
- ❖ Audio output of approximately one second every time the user enters an invalid input (buzzer).

2.12.1.4 – Timing

Timing is not an issue for the serial communication module. As a result the receive and transmit interrupts were set to low priority not to interfere with the PWM module.

2.12.1.5 – Failure modes

In the case where the serial connection fails (i.e. the keyboard stops responding to input from the user, or constant display of strings starts occurring), it indicated the occurrence of variable corruption in the menu module and the program will need to be restarted.

2.12.2 - Non-Functional (Quality of Service) Requirements

The serial connecting port should be plugged in and screwed before the device is turned on. Make sure the connection is not loose to prevent loss of data from happening.

2.12.2.1 – Interfaces

The interface is designed for a user who has sensory perception of the eyes. The buttons are not designed for blind users. Furthermore unlike in keypad mode, a blind user would not be able to navigate through the menus with the help of the buzzer but would only be notified when he entered an invalid input with the same buzzer.

2.12.2.3 - Design Constraints

Displayed strings were kept as short as possible due to memory constraints

2.12.3 - Conceptual Design: Serial Module.

When designing the serial interface it became obvious early in the design process that the serial communication module had to be entirely interrupt driven. The problem with Polling and waiting for the TXREG flag to set is that the CPU cannot do anything else at the time. This is not much of a problem for small programs, but due to the complexity and the number of modules to be integrated for this project, it was crucial to design an interface that could transmit and receive characters without affecting the overall behavior of the device, such as interfering with the PWM or sensor Module.

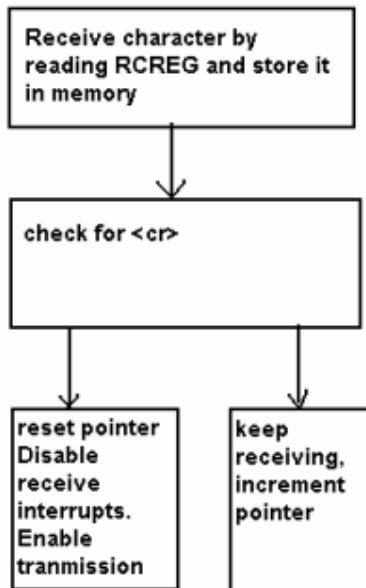
It was also very important to design a serial communication system that was as modular as possible. This is the reason why the transmission was carried out using the functions `tx232C` and `tx232C_RAM`, which are passed the pointer to the start of the string to be transmitted. Data memory usage was a big concern during the development of the program. The PIC board we were provided with had 1024 bytes of Ram, of which half was used by the stack. Therefore we were only given about 600 bytes of RAM to write the program with. As a result we decided to store most of our strings in program memory and use a buffer to store the changing data. We therefore implemented two different transmit functions, one which transmits strings stored in program memory (mainly information strings), and one that transmits strings stored in data memory such as the user input or readings from the device. This is quite an unusual way of implementing a serial communication system, as the most common way would be to get the strings from program memory and transfer them to data memory using table read and table write operations. We understand the problem in our design, but ran out of time to go back and make the necessary modifications to it.

2.12.3.1 - Flow diagram

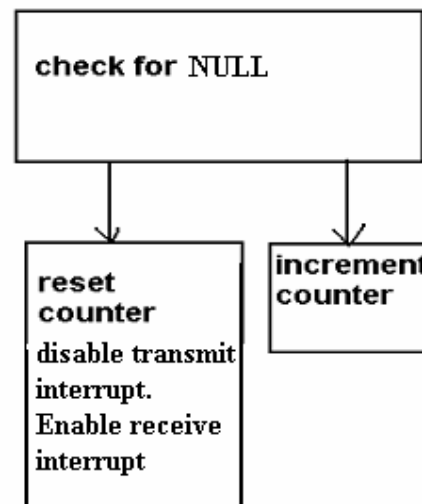
The diagram below shows the basic principle behind the serial interface module.

In the final program, the keypad had priority over the serial interface and both the receive and transmit interrupts were disabled and were only turned on from the keypad when switching to factory mode or use remote mode. When doing so, the transmit interrupt is first enabled to display the command menus. Because the interface is entirely interrupt driven, the transmit function sends each character one after another until it reaches the NULL terminator at the end of the string. Once that happens the transmit interrupt is masked and the receive interrupt is enabled so that the user can start inputting values to control the device.

rx232_isr



tx232C and tx232C_RAM



Because the programmer has no possible influence on the state of the transmit and receive flags whatsoever due to the fact that they are cleared automatically in hardware, the transmit and receive interrupts were set to low priority and inserted at the very bottom of our low interrupt handler.

2.12.3.2 - Testing and observation

During testing we encountered recurrent problems when sending a carriage return followed by a line feed through the transmit line. Indeed the last character appeared not to be sent. It turned out that the reason why this was happening was because we would disable the

transmission line before making sure that the last character to be sent had properly been sent through the serial transmit line. To solve this problem, it was made sure that the TXREG has transferred the data to the TSR register and then that the TSR has sent the data to the serial pin before the transmit line was disabled using the simple line of code.

```
while(TXSTAbits.TRMT == 0);  
TXSTAbits.TXEN = OFF;
```

2.5.3.3 - Buffer overflow.

An important thing to account for when dealing with strings of characters is the size of the buffer. Its size is defined at the start of the program and the programmer needs to account for the possibility that it might overflow if the user enters a string of too many characters. An easy way to do this is to check for the value of the counter we increment each time data is stored in the array. If its value is equal to the size of the buffer, we stop storing the new incoming data in the buffer.

2.13 – Module Requirements: User Interface Module (Menu)

The operational scenarios considered place certain requirements on the Menu User Interface Module and on the sub systems that comprise it.

2.13.1 - Functional Requirements

This section describes the functional requirements of the Menu Module – those requirements that must be met if the module (and system) is to function correctly.

2.13.1.1 – Inputs

The menu module receives no external inputs and is completely controlled internally by triggers from the keypad module.

These internal triggers are:

- ❖ The new keypad entry
- ❖ The flag indicating new keypad entry

2.13.1.2 – Processes

The menu module is contained in the program while(1) loop. It will execute if and only if the flag indicating new keypad entry is set. Once this flag is set, three processes are undergone in succession:

1. The current menu state is updated (e.g. the enter key is pressed, and the menu transitions to the next level).
2. If this update is triggered, then the strings to be sent to the LCD module are then updated by a second function.
3. If these strings have variable values (such as the display min/max range, or the entry of numerical data), the current strings are edited with the variable data.

Note: The sending of the strings to the LCD is taken care of in the LCD module.

2.13.1.3 – Outputs

The outputs of the menu module are threefold:

- ❖ An internal output within the menu module is the flag to indicate that the LCD must be updated. This flag is set at the end of process 1 and triggered at the beginning of process 2. (processes shown above)
- ❖ After the three processes have executed, the new strings to be sent to both lines of the LCD are produced and stored in a globally accessible location.
- ❖ Internal outputs from process 1 include the new menu state (defined by two variables: menu_level (which menu) and menu_depth (how far into the menu you are)).

2.13.1.4 – Timing

No timing is associated with the menu module. It is executed once a rising edge interrupt triggers the new keypad input flag and once the main while loop enters the 'if' statement for the LCD update.

2.13.1.5 – Failure modes

If a menu variable becomes corrupted, the keypad will still sound, but the menu would not update. In this scenario the menu module has failed, and a manual restart of the entire system must occur.

This failure happens infrequently, but when it does it is usually due to the keypad interrupt interfering with and LCD update.

2.13.2 - Non-Functional (Quality of Service) Requirements

The menu module simply restricts users to the temperatures outlined for the use of the DST. Outside these ranges, the PIC may perform sub-optimally and the internal menu module may experience corruption of variables.

2.13.2.1 – Performance

As mentioned before, no new requirements are set for quality performance of the menu module.

2.13.2.2 – Interfaces

The menu module has a completely internal interface, so the quality of the interface is only constrained by the performance of the PIC.

2.13.2.3 - Design Constraints

The design constraints of the menu module were simply limited data memory and program memory. The strings used for the menu module were stored in program memory, whereas the update-able strings to be sent to the LCD were stored in data memory. In this way, the extremely limited data memory was conserved, whereas the significantly larger storage capacity of program memory was employed for storage.

2.14.3 - Conceptual Design: Menu Module

2.14.3.1 - Initial Menu Design – Menus

The initial design of the menu module involved a menu which could perform the following tasks with outlined depths below.

- ❖ Welcome Screen

This screen was to display “Welcome to Team Cinder Death Star Tracker”, but as this string was too long, it was contracted into “Welcome to Team Cinder DST”. This welcome screen was then followed by a string indicating a choice of entering the menu mode, or starting the scanning.

- ❖ Scanning Screen

If the scanning mode was selected, this screen was to display the status of the object as either scanning or tracking (Not the current location estimates), with a press of any key triggering an entry to the menu interface.

- ❖ Menu Screens:

- ❖ Show Target Status

Shows the targets current location as (elevation, azimuth, range) in degrees, degrees and mm respectively.

- ❖ Show Temperature

Shows the temperature in degrees Celsius.

Both of these above menus have only two depths. The first is simply the menu selection displaying their names. The second is an actual display of the current values of the status or the temperature.

- ❖ Set Minimum Azimuth

- ❖ Set Maximum Azimuth

- ❖ Set Minimum Elevation

- ❖ Set Maximum Elevation

These above four menus were to perform the prescribed task of setting the minimum/maximum azimuth/elevation. The menu has four levels:

1. The menu selection screen (displaying Set Min Az etc.)
2. The value entry screen (displaying the current sign of the entry + or -, as well as its current value. E.g. Azimuth: + 45)
3. A confirmation screen to make sure the selected values were what the user wants.
4. An error screen if the value was outside prescribed ranges, or contradictory in some way (i.e. max < min).

These two methods of depths (menu-display and menu-entry-confirmation-error) are used exclusively for the rest of the menu and will be signed as either 2-depth or 4-depth menus respectively.

- ❖ Set Minimum Range
- ❖ Set Maximum Range

These menus are exactly the same as the aforementioned four setting menus as a 4-depth menu with one minor difference. The positive and negative switching is disabled, as the minimum and maximum ranges must be set as a positive number.

- ❖ Goto Azimuth
- ❖ Goto Elevation

These menus are a standard 4-depth, with a check to see if the desired angle is within the limits of the maximum and minimum ranges defined by the first functions.

- ❖ Set Factory Mode

This menu is a 4-depth menu with a check to see if the entered value is equal to the code. For simplicity the code to be used would just be either a + or – value followed by a three digit number. (In our current case this would be -747). This should be sufficient to deter a normal user from entering factory mode (2000 combinations).

- ❖ Set User Remote Mode

This menu is a 2-depth menu with a menu-confirm depth.

Both of the above modes relinquishes the control of the keypad, and gives it to the serial port in either the factory or remote modes. The keypad interrupt is stopped from firing during the conversion, so they keypad will not make any more beeps, and will be ineffectual in changing the display on the LCD.

Notably, even if the keypad was to work, it could not change the state by the way the software was designed.

- ❖ Exit Menu

This menu is a 2-depth menu-confirm, with 2 exit states. If the object is being scanned or tracked, the menu will exit to the scanning screen. Otherwise it will exit to the second welcome screen, allowing the user to start scanning or re-enter the menu.

This design was planned and implemented (which will be discussed below), however it was found that with added functionality (such as the 3d scanning) new menu options needed to be added and changed.

The menu options that needed slight changes were:

- ❖ Set Minimum Azimuth
- ❖ Set Maximum Azimuth
- ❖ Set Minimum Elevation

❖ Set Maximum Elevation

Due to the way the maximum and minimum of elevation and azimuth was designed in the serial interface, the maxima and minima of each can only be positive and negative numbers (including 0) respectively. This required the menu to have a small modification to disallow incorrectly signed numbers. However, this removed the conflict between the max < min scenario and made the code equally complex as before.

❖ Show Target Status

It was decided that the target status will now have three different states. Instead of always displaying the current azimuth and elevation of the tracking device, and the distance of whatever was being measured it would have three states:

1. Whilst not active, the display will be: Not Active...
2. Whilst scanning, the display will be: Scanning...
3. Whilst tracking, the display will be the triple of elevation, azimuth and range.

New menu options:

- ❖ Show Min/Max Azimuth
- ❖ Show Min/Max Elevation
- ❖ Show Min/Max Range

As the azimuth, elevation and range minima and maxima were not easily accessible; it was decided to implement a function which allowed them to be read. This acted as both a user interface, and an error detection device.

This was implemented as a 2-depth menu-display.

❖ Enable/Disable Scanning/Tracking

The menu design was clumsy for enabling and disabling scanning and tracking, so a new menu option was built in (displaying the current state as well). The initial menu design was kept as a hangover; it did not impede the user in any way, so there was no necessity to remove it.

This was implemented as a 2-depth menu-confirm.

❖ Options Menu

Once 3d tracking was designed, it was decided to implement it in an options menu. The options menu acted as a 4-depth modified menu: menu-3d-buzzer-shutdown. The menu was navigated by pressing the enter key, and 1 was depressed (with no confirmation) on the menu of choice to enable or disable any of two things:

- ❖ 3D Tracking
- ❖ The keypad buzzer

The third options menu was the shutdown procedure which was designed so that the product shuts down in a well defined way. The procedure designated that the serial cable be removed before the power to the device was turned off. Notably, the device would not be able to restart without the power being restarted (as the keypad was disabled on the final screen).

2.14.3.1 - Initial Menu Design

2.14.3.1.1 - Menu Selection

In order to conserve space on the PIC the menus were reused when necessary. Such as using the same error messages whenever each error occurs. This involves designing a menu selector which picks the required strings for each line to be sent to the LCD.

Each menu was given a number, outlined in Appendix B.

And each string was given a number (as an index in the array) (Appendix B). Note that the number 0 was forgotten (by designer fault), so an arbitrary "blank" string was put in its place. (This used up an extra 17 redundant bytes of program memory, but this was not an issue for us once we integrated fully).

Note that the strings 15 and 23 are the same. However, they were to be used separately for elevation/azimuth vs. range/factory password until it was decided that four characters was a sufficient standard for entering an angle, in case the system was refined for larger angles.

And for each menu level and depth we have a lookup table which defines the string number based on the menu level, depth and the line number. (Note that some variables have multiple values depending on external flags such as tracking and the state of the keypad buzzer). Shown in Appendix C.

In order to reduce the size of the code, most of the common cases were combined together in a large switch statement. However it was not tested whether the mathematics used in the menus were useful in reducing the size of the code. However, since they were implemented successfully before integration, there was no need to change them later.

2.14.3.1.2 - String Updating

String updating was performed once the string number was extracted from the lookup table above. Both string numbers were extracted for each line, and stored in a temporary variable. This was only performed if the menu changed states. Once the string numbers were extracted, the strings were copied to the temporary locations, and the temporary strings were sent to the LCD.

2.14.3.1.3 - Initial Menu Design – Variable Strings

However, some of these strings had dynamic variables put into them – such as target locations and keypad input numbers. Within the string update function, once the strings have been updated, it was checked whether they were the variable strings. These were then updated with the required values. The values were determined from the string number as well as the menu state within the string updating function. I.e. if the min/max of azimuth was to be found, it was not directly inputted into the function that the azimuth was to be found, rather it was determined from the new menu state that azimuth needed to be displayed.

These required values had to be stored in some form of array, which was the keypad storage.

2.14.3.1.4 - Storing Keypad Values

Whenever a number was being processed by the keypad/LCD interface (when a number was being entered for min/max/goto azimuth/elevation/range) the number was stored in two ways:

1. The numeric value was stored in an unsigned int
2. The characters to be sent to the LCD were stored in a 4 length array, containing not their values but the character representations of the integers and +/-.

Consequently, a function was made which took the new keypad input, and globally altered the values of these two variables.

2.14.3.1.5 - Menu Transitions

The menu transitions were written on-the-fly with a basis of simple rules:

- ❖ From initial menus (depth = 0), the up and down rocker switch generally changes the menu level by ± 1 . Whenever enter was pressed, it simply changed the menu level to 1.
- ❖ If any button was pressed from the final menu, it triggered depth = 0.
- ❖ Each menu generally incremented the depth by 1 every time enter was pressed.
- ❖ Once the 3rd menu was entered (the confirm stage) the error checking generally occurs here. The result of the check triggers either a change, or depth = 0 or an error and depth = 3 with no change.
- ❖ Other menus would be dealt with individually (such as the options menu, which had a different structure due to its writing at such a late stage).

2.14.3.1.5 - Logbook Entries

These are some of the pages from the design of the menu. They depict lookup tables as well as the mathematics to define the strings and the strings themselves.

Initial write-up of the lookup table shown in Appendix D:

Initial string definition. Note how string 23 and string 15 were initially different to indicate their different usages. (Appendix D)

Initial mathematics for the string definitions. Note the use of bit-shifts wherever possible to reduce computation time.

2.14.3.1.6 - Functional Approach

In pseudo code, the menu was to trigger if a new keypad entry was received and then proceed as follows:

Function: Uptade_LCD

Begin:

Update the menu state

If (Strings need updating)

update Strings

display strings on LCD

End

2.15 - Module Requirements: Keypad Module

The operational scenarios considered place certain requirements on the Keypad User Interface Module and on the sub systems that comprise it.

2.15.1 - Functional Requirements

This section describes the functional requirements of the Keypad Module – those requirements that must be met if the module (and system) is to function correctly.

2.15.1.1 – Inputs

The keypad module has only two forms of inputs. The keypad itself and the rocker switch located to the left of the keypad. Both of these are located on the user interface enclosure.

These inputs are connected to a keypad controller decoder chip (MM74C922) which then connects directly to the PIC.

2.15.1.2 – Processes

The keypad module performs a simple interrupt driven routine which receives a rising edge interrupt (which is triggered by user input on the rocker switch or the keypad through the keypad decoder chip) and then processes the current value entered on the keypad. The current value is read off the keypad decoder chip through a series of 4 pins which output values between 0 and 15.

2.15.1.3 – Outputs

The outputs of the keypad module are threefold:

- ❖ The value entered on the keypad itself (or the rocker switch). This value is decoded to equal: 0-9 for the digits 0-9 and [* , # , UP , DOWN] = [10, 11, 12, 13] for the other 4 input buttons. It is stored in a globally accessible location.
- ❖ The flag to trigger a menu update due to keypad inputs. It is stored in a globally accessible location.
- ❖ Audio output of a short beep every time a key or the rocker switch is pressed. Note: This can be turned on or off within the menu system.

2.15.1.4 – Timing

No timing is associated with the keypad module, however the priority of the rising edge interrupt is very low, thus not impeding the time dependent interrupt routines (such as the PWM module).

2.15.1.5 – Failure modes

In the case where the keypad fails (i.e. the keypad stops responding to input from the user) this is clearly defined by a lack of a beep sound. If the keypad is still beeping, but no change in the LCD occurs, this means variable corruption in the menu module has occurred (and the program will need to be restarted).

2.15.2 - Non-Functional (Quality of Service) Requirements

All wires should be kept far away from all oscillating voltage sources (such as 240V mains wires). These wires can induce currents in the system that will stop everything working correctly.

Before the keypad interface is used, it must be cleaned free of all moisture (ideally before the system is turned on). Consequently the digits of the users must also be free of moisture before use of the keypad ensues.

2.15.2.1 – Performance

The keypad and rocker switch are required to be kept at standard room temperatures for optimal performance. Outside these temperatures the keypad may perform sub-optimally or not at all.

2.15.2.2 – Interfaces

The interface is designed for an end user who has sensory perception of the eyes. The buttons are not designed for blind users. However, as the format of the keys is known, a blind user can memorise the location of all buttons, and through the use of the buzzer (noting when a keypad press occurs) can navigate the menus from memory.

2.15.2.3 - Design Constraints

The design was limited to a total of 16 possible buttons for the given keypad decoder. The design of the menu required only the use of 14 buttons, and thus was within the constraints of the menus.

2.16 - Conceptual Design: Keypad Module

The keypad module was designed with the menu module in mind. The following keys were deemed necessary for the operation of the menu:

- ❖ The numbers 0 through 9
- ❖ A clear button
- ❖ An enter button
- ❖ Buttons to cycle through the menus (essentially an up/down combination)
- ❖ Buttons to indicate positive or negative angles.

Given the constraint of a 4x3 keypad the following button alignments were to be used:

- ❖ 0 through 9 acted as the numbers they described.
- ❖ * was the clear button
- ❖ # was the enter button
- ❖ 4 and 6 were the buttons to cycle through the menus.

However, this left no buttons to indicate the sign of the angles. A solution was to incorporate a secondary menu into the designation of the angle to decide positive or negative. However, this would lead to an obtuse user interface. An alternative solution (which was the one proposed) used a rocker switch for the cycling through menus, as well as to indicate positive or negative angles during their input.

Thus the final button alignments were decided as follows:

- ❖ 0 through 9 acts as the numbers they describe.
- ❖ * is the clear button
- ❖ # is the enter button

- ❖ UP and DOWN on the rocker switch cycle through the menus.
- ❖ UP and DOWN on the rocker switch indicate positive and negative angles respectively.

The software for the keypad was initially designed as a simple rising edge interrupt routine (triggering off the keypad decoder chip) which performed the following tasks:

- ❖ The input to be read and decoded
- ❖ The setting of a menu update flag to update the LCD

However, this simple design left something to be desired, as for most button presses, the LCD need not be updated. Instead a new keypad entry flag was set to note that the keypad has been pressed. This in turn triggered the LCD update module, but the LCD would only update if the keypad input deemed it necessary (such as cycling menus or inputting new numbers, rather than pressing a number while cycling through menus which would not trigger an update).

Further into the design process, it was decided that a third task would be performed: A beep would sound whenever a key was pressed. This allows even more feedback to the user, as they would necessarily know if a key had been pressed.

As an addition in the late stages of design, the options menu was introduced and in it the sound produced by the keypad could be turned on or off.

2.17 - Module Requirements: 3D Tracking and Scanning

The operational scenarios considered place certain requirements on the 3D Tracking and Scanning Module and on the sub systems that comprise it.

2.17.1 - Functional Requirements

This section describes the functional requirements of the 3D Tracking and Scanning Module – those requirements that must be met if the module (and system) is to function correctly.

2.17.1.1 – Inputs

For the 3D Tracking and Scanning module to execute, the 3D tracking flag must be set (by default). The other inputs to the module include:

- ❖ The current 3D tracking state (i.e. how far into the algorithm it is)
- ❖ The lookup table for the next servo positioning (for both Tracking and Scanning).
- ❖ The current 3D tracking multiplier (the angle by which the servos increment each tracking cycle).
- ❖ The depth of the tracking algorithm (how many cycles it executes before it deems the object lost).
- ❖ The current 3D scanning multiplier (the angle by which the servos increment each tracking cycle).
- ❖ The depth of the scanning algorithm (how many cycles it executes before the scanning algorithm repeats itself).

2.17.1.2 – Process

The 3D Tracking and Scanning module takes the current 3D tracking state (represented as an index), and changes the position of the servos based on the current location of the servos.

Due to the construction of the tracking and scanning algorithms, both elevation and azimuth need only be incremented by a constant amount (being either 0 or \pm the tracking/scanning multiplier).

By construction, if the new positions on the tracking/scanning algorithm are outside the range of the system, it will ignore those commands and move further through the list until it finds a valid move to make, then the algorithm continues on its way.

So although boundary behaviour can be deemed as somewhat erratic, it does not violate the boundaries of the system and will continue tracking/scanning as required.

2.17.1.3 – Outputs

The outputs from the 3D tracking and scanning module are just the new servo positions (i.e. azimuth and elevation). These calculations are performed, and the servos move to a new location.

2.17.1.4 – Timing

Unlike in the 2D Tracking and Scanning module, there is no artificial delay created within the module. This is due to a construction error in the algorithm, which yielded positive results.

The algorithm was initially designed with a wait routine in mind, however at time of initial testing; the wait routine was not implemented so that the module could be tested roughly.

Conveniently enough the algorithm displayed incredibly promising results, tracking a ball very quickly (much better than the current 2D algorithm). Consequently it was decided that the wait routine would not be included in the new algorithm.

The justification for this lack of inclusion is as follows: When the sensors detect a target, the tracking routine is exited and begins its 'wait there' routine. However, if the servos have moved since last detection, the module will find no target again and re-enter the tracking routine. This will eventually find the target (as usually the servos will not move again before the measurement, and hence detection, is made).

2.17.1.5 - Failure modes

If the tracking routine fails to pick up the object, it immediately exits into the scanning routine. There are no inbuilt failure detection modes within the module (such as detection of actual servo location etc.) and it will continue to operate even if the hardware is faulty.

2.17.2 - Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

2.17.2.1 – Performance

Performance is controlled almost exclusively by the performance of the hardware. Also, if the keypad is being used often, the performance of the tracking algorithm can be noticeably reduced (as the keypad is taking up sufficient time to reduce the speed at which the servos are changed).

2.17.2.2 – Interfaces

The interface of the 3D tracking routine is purely software and will thus not be analysed to any significant extent.

2.17.2.3 - Design Constraints

The design of the tracking algorithm has no real constraints apart from speed of execution (for a tracking algorithm to be effective it must be fast).

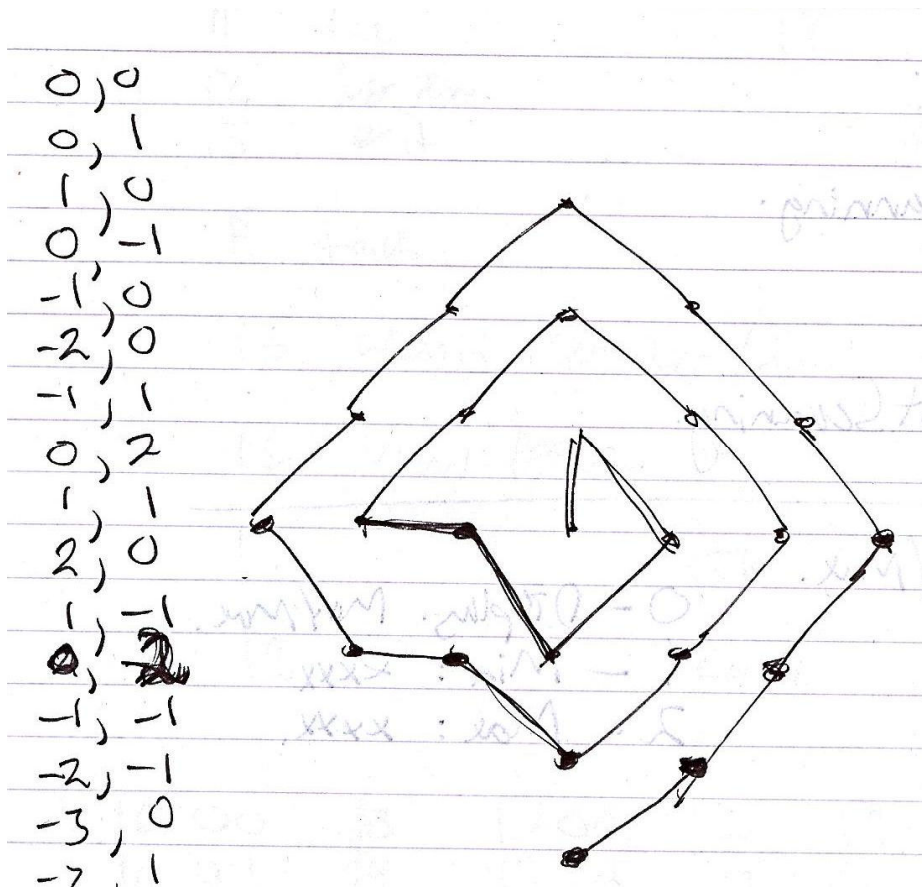
Consequently, the simplest algorithm for the job was chosen (a simple spiral outwards).

2.18 - Conceptual Design: 3D Tracking and Scanning Module

Given the construction of the 2D Tracking and Scanning Modules, the only significant thing to write is the new movement algorithm. I.e. define a new path over which to scan once the target has been lost.

Once this had been written it was extended into a simple scanning routine which 'tracked' for longer, and if the object never found just repeated the spiral 'tracking' procedure.

The algorithm was constructed from the following spiral diagram (which is an excerpt from a logbook):



The numbers represent the Cartesian coordinates for the tracking algorithm (before the multiplier application), and the diagram is in angular space (i.e. azimuth angle vs. elevation angle). Note that this diagram is NOT what the tracking algorithm appears to do if the servos had a pen attached drawing on a spherical shell of paper, but is a close approximation (due to the way the angles add, they're not the same as Cartesian coordinates).

Note that this algorithm never takes more than one step in either direction of azimuth or elevation. This makes its implementation vastly simpler as you need not worry about how far to move in any direction given that direction.

Encoding the data into a single vector array was done through excel where the relative changes in coordinates were calculated (0 for same, 1 for positive, 2 for negative) and then the values in azimuth were multiplied by four and added to the values for elevation.

A sample input would be: (positive azimuth, negative elevation) = (1,2)

This is then taken to: $1*4 + 2 = 6 = 0b0000aaee = 0b00000110$

The reason for using this condensation of data was to conserve space. Conceivably for a more complex algorithm many more states could be stored using only one nibble for each of azimuth and elevation.

The tracking algorithm then follows this path for 40 steps (arbitrary, but a good value to keep a fairly fast moving target). If the object is lost it begins scanning again. If the object is found, the tracking algorithm is not reset (alas, an oversight at time of coding) and the servos will remain stationary until the object is lost again.

3. USER INTERFACE DESIGN

3.1 - Classes of User

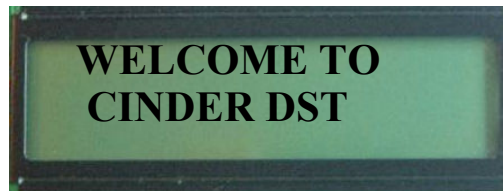
The DST comprises of two classes of user defined as; Factory mode and User mode. User mode has 2 subsystems, User Local and User Remote mode.

3.2 - System Functionality

3.2.1 – User Local Mode

- ❖ After sensors have been moved by the servos to the zero position, LCD will display:

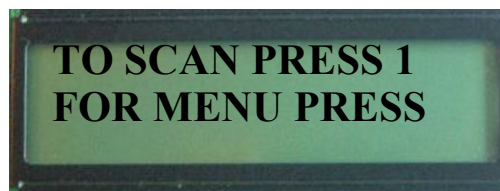
SCREEN 1



* DST = Death Star Tracker *

- ❖ Pressing any key will result in the following menu screen:

SCREEN 2

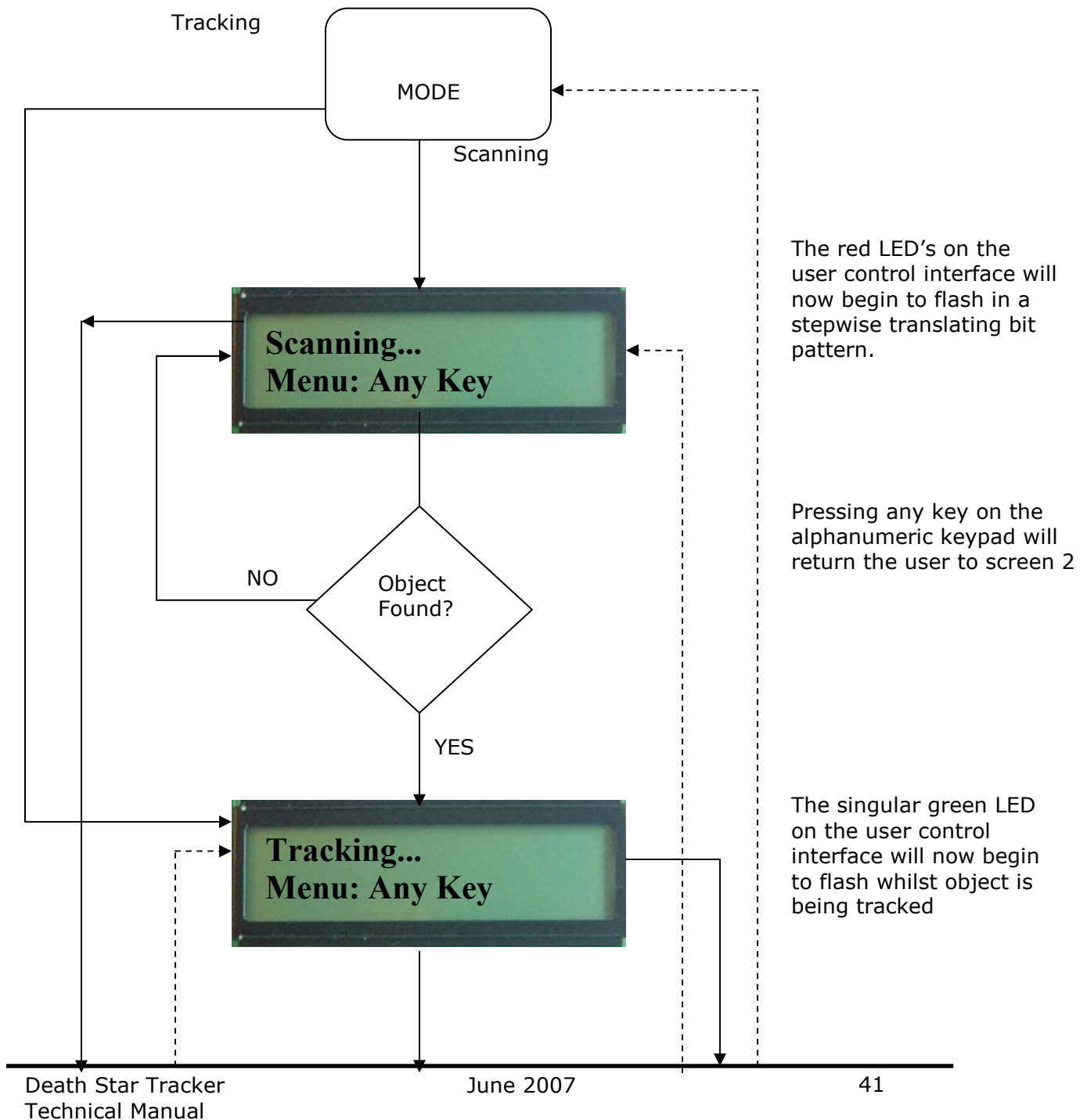


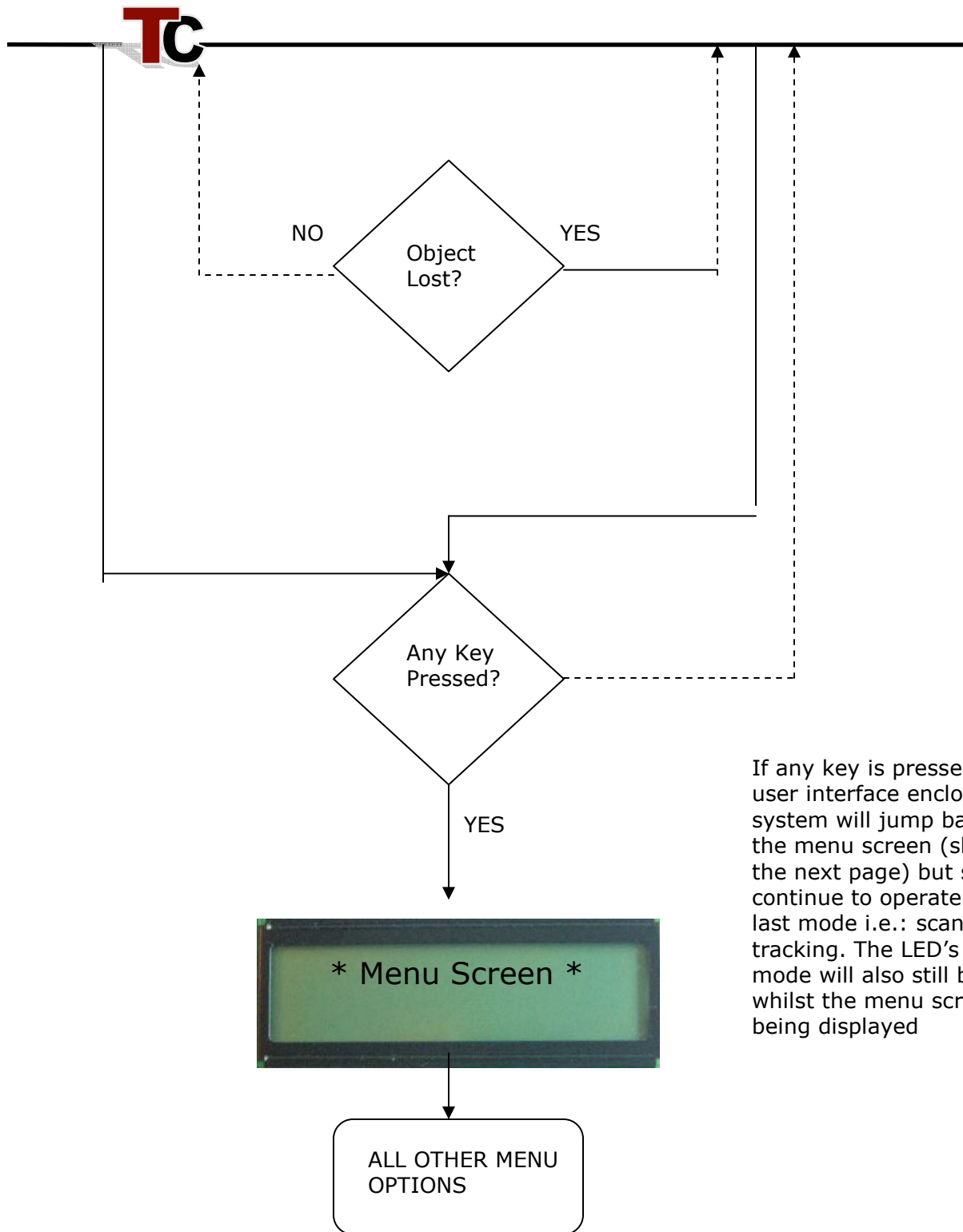
ALL INNCORRCT ENTRIES WILL BE ACCPOMPANIED BY AN ERROR SCREEN AS WELL AS A SHORT BEEP FROM THE BUZZER

* Buzzer located inside the user control interface enclosure *

3.2.1.1 – Option 1: Scan

As soon as the number one is pressed on the keypad, the system will enter a sweeping scanning sequence in a horizontal direction mode from position [0 0] (azimuth and elevation respectively).





If any key is pressed on the user interface enclosure, the system will jump back top the menu screen (shown on the next page) but still continue to operate in its last mode i.e.: scanning or tracking. The LED's for each mode will also still be active whilst the menu screen is being displayed

3.2.1.2 – Option 2: Menu

Pressing 2 on the keypad will trigger a menu system operated by the keypad and the rocker switch.



The rocker switch controls the menu system by scrolling up and down the options as shown to the right. Depressing the upper switch scrolls up and subsequently, depressing the down switch scrolls down in the menu options.

As each option is triggered to scroll up and down, the lines displayed on the LCD will also move up and down respectively.

In the Options menu, the user can select from the following parameters;

1. Disable/Enable 3d tracking
2. Disable buzzer on Keypad
3. Enable Shutdown

Show the Current Target Status
Show the Current Temperature
Set Minimum Azimuth

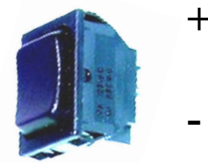


ELEVATION MAXIMUM
Set Minimum Range
Set Maximum Range
Go to Azimuth
Go to Elevation
Change mode to Factory
Change mode to User remote
Display Min/Max Azimuth
Display Min/Max Elevation
Display Min/Max Range
Start Scanning...
Options Menu
Exit Menu

MENU SCREEN

If a number is pressed in this menu mode, e.g.: 4, the following screen will appear.

SCREEN 3

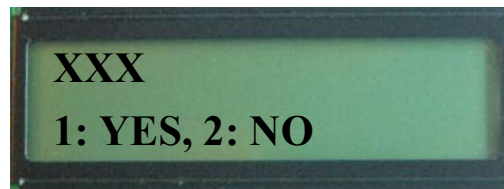


The user can then enter the maximum desired azimuth value that the system will operated under using the alphanumeric keypad.

- ❖ Negative values can be entered using the rocker switch with Up switch being positive and Down switch being negative.
- ❖ The default sign is positive.
- ❖ The star key (*) clears the screen incase of incorrect user
- ❖ Values are confirmed using the Hash key (#) as enter

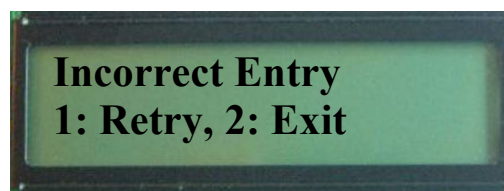
The next stage is a confirmation screen for all menu options as shown:

SCREEN 4



If the user presses 1 to confirm the entered value, the system will accept the input and move on to the next stage, validation. If the input is beyond parameters, the following error screen will be displayed.

SCREEN 5



Hitting retry takes the user
back to screen 3
Exit goes back to Menu screen

If the correct input parameters are obeyed and the value is accepted, then the system will return to the Menu Screen with the new value for the variable stored.

At all times during the menu mode, the scanning and tracking system will be disabled (LED's will be inactive)

The rocker switch will automatically switch between modes of use from one input depth level to the next. I.e.: negative / positive in one option and up / down in the next.

The next stage or product operation is to enter USER_REMOTE mode or FACTORY_MODE (requires a password – please see your nearest team cinder technician) which are both accessible through the menu screen stage of operation.

3.2.2 – User Remote Mode

User remote mode disables all hardware input on the user control interface and enables HyperTerminal. User Remote mode can only be entered from the menu screen stage of operation in User Local mode.

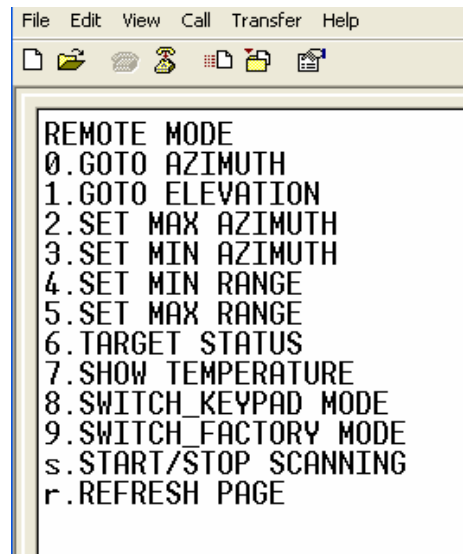
After entering (Hash key #) option [12 = set_user_remote] from the menu screen, the LCD will display



Selecting 2: NO
will once again return the
user to the menu screen

If (1: YES) is pressed, the user interface elements will deactivate and the system will switch to keyboard and computer screen input output respectively use using HyperTerminal running at 9600 Baud.

When HyperTerminal is opened, the user will be able to select the following options using the keyboard buttons;

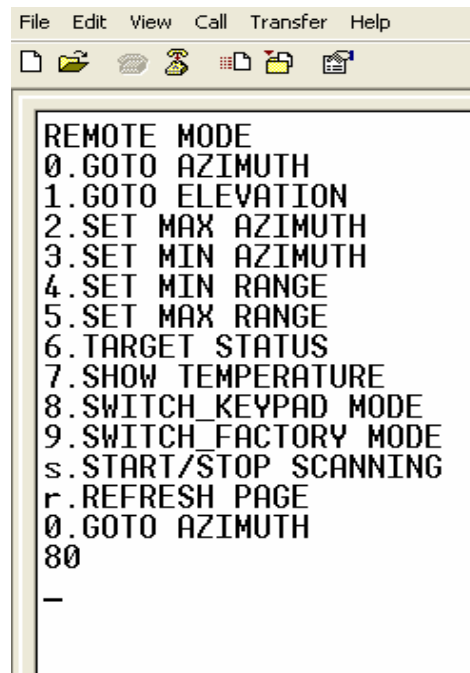


```
File Edit View Call Transfer Help
[Icons]

REMOTE MODE
0.GOTO AZIMUTH
1.GOTO ELEVATION
2.SET MAX AZIMUTH
3.SET MIN AZIMUTH
4.SET MIN RANGE
5.SET MAX RANGE
6.TARGET STATUS
7.SHOW TEMPERATURE
8.SWITCH_KEYPAD MODE
9.SWITCH_FACTORY MODE
s.START/STOP SCANNING
r.REFRESH PAGE
```

In the same way as the keyboard, the user can select each option which will bring up a new menu


For example; pressing 0 then 80 then enter will command the servos to move to azimuth position 80. The system will then display the variable changed and the new value as shown below.



```
File Edit View Call Transfer Help
[Icons]

REMOTE MODE
0.GOTO AZIMUTH
1.GOTO ELEVATION
2.SET MAX AZIMUTH
3.SET MIN AZIMUTH
4.SET MIN RANGE
5.SET MAX RANGE
6.TARGET STATUS
7.SHOW TEMPERATURE
8.SWITCH_KEYPAD MODE
9.SWITCH_FACTORY MODE
s.START/STOP SCANNING
r.REFRESH PAGE
0.GOTO AZIMUTH
80
-
```

All incorrect entries will result in the following message being displayed as well as the buzzer sounding a sharp beep.

 Invalid Input
—

- ❖ All commands in User Remote mode and Factory mode must be accompanied by the enter key.
- ❖ The LED's on the user interface enclosure will also function as a tracking and scanning visual aid in this mode of system operation
- ❖ Also similar to the user Local mode, the user can change to Factory mode or back to User Local mode by pressing the appropriate menu key and hitting enter.
- ❖ If the user opts to return to User Local mode, HyperTerminal will disable and will only be re-enabled with the correct sequence of events as described in section 3.6.2

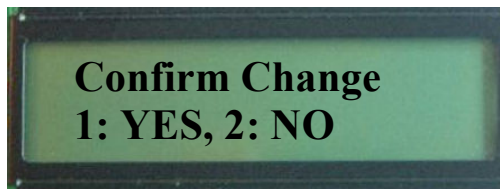
3.2.3 – Factory Mode

Entering factory mode requires the user to pass a security clearance in the form of a 4-digit password.

From the factory mode the user will be able to change the calibration of IR and US sensors, temperature sensor and azimuth / elevation parameters.

3.2.3.1 - Entering factory Mode from User Local Mode

From the menu screen, pressing option (14: Factory Mode) will result in the following screen;

 Confirm Change
1: YES, 2: NO

And when option 1: YES is pressed on the keypad,

The following security screen will be displayed;

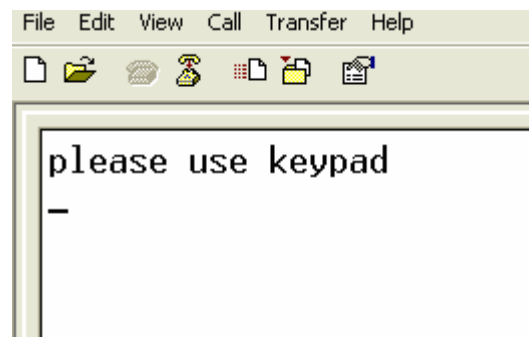


If the wrong password is entered, then the system will display the error message shown previous in section 3.7.1.2 (screen 5)

If the correct password has been entered, the system will automatically disable the user control interface enclosure and jump into factory mode using the keyboard as input and HyperTerminal as output.

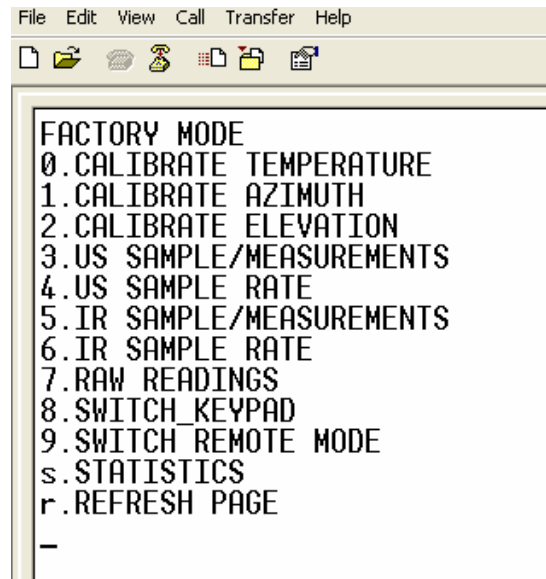
3.2.3.2 - Entering factory Mode from User Remote Mode

Selecting option 9 from the HyperTerminal menu, then entering the 4 digit password and enter will bring up the HyperTerm disable window and the user must now use the Keypad



If the wrong entry is typed in, the system displays the error message shown in section 3.7.2 and gives the user the option to keep trying until clearance passed or other functionality required.

Factory Mode itself is an extension of both user modes; providing calibration features as shown below:



Each option is bound by input parameters (given in section 3.6) and an invalid input will result in an error display as shown in section 3.7.2.

After an input has been accepted by the user, HyperTerminal stores the value in its appropriate variable name and returns the user to the Factory Mode menu screen.

As standard in all modes, switching can only be triggered from external menu screen with no password required to exit Factory Mode.

The LED's on the user control interface will also function for scanning and tracking in Factory mode.

In all modes of operation, the buzzer will sound to notify the user that a key has been successfully registered by the system. The buzzer will also emit a longer beep for all error and invalid input messages encountered.

The Buzzer can be turned off at any time during product operation through the Options Menu.

Other options menu sub sections are:

- ❖ Stop 3D tracking (to begin 2D tracking mode)
- ❖ System Shutdown – whereby the servos will move to zero position, keypad will de-initialize and the system can be safely turned off

3.3 – Shut-down Sequence

- ❖ When the system needs to be shutdown, the servo's should first be instructed to stop (hash key using the keypad and stop scanning using HyperTerminal)
- ❖ The red switch should then be toggled off to kill power to the entire circuit. The blue power LED will turn off
- ❖ The internal PIC minimal board will always be running unless the SLA battery is manually disconnected; thus in normal operation, the program can be easily restarted after stoppage by toggling the red power switch.
- ❖ Team Cinder **STONGLY RECOMMENDS** that the user **NEVER** disconnects the SLA battery. For prolonged product use please talk to your nearest Team Cinder Technician.

3.4 - Interface Design: User Class Y

3.4.1 - User Inputs and Outputs

In User Local mode, all Inputs are commanded by the buttons on the User Interface Enclosure.

+



-



Entering Values for azimuth, elevation, tracking and scanning.

* = Clear key

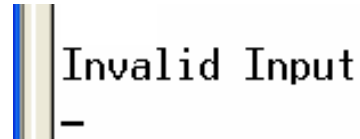
= Enter Key

In User Remote and Factory Mode, all inputs are handled through the keyboard.

All data inputted and the resulting changes to the system will be displayed either on the LCD or in HyperTerminal. The triggering of a tracking sequence will initialise 6 red LED's to flash in a continuous stepwise translating bit pattern; and once the object has been found, the green LED will begin to flash. The LED's will activate for all modes of operation.

3.4.2 - Input Validation and Error Trapping

In all modes of system operation, an invalid user input is always returned with the output.



The user is then returned to the previous phase of functionality as outlined in section 3.7

4. HARDWARE DESIGN

4.1 - Scope of the DST System Hardware

The hardware for the DST which has been custom designed includes:

- ❖ Handheld controller
- ❖ Base unit enclosure
- ❖ Battery charger circuit
- ❖ Power supply circuit
- ❖ LCD contrast circuit
- ❖ Buzzer
- ❖ LED Scan circuit
- ❖ Keypad Decoder circuit

The hardware for the DST which has not been custom designed includes:

- ❖ PIC Minimal Board
- ❖ Servos
- ❖ Ultrasonic Sensor
- ❖ IR Sensors
- ❖ Temperature Sensors

4.2 - Hardware Design

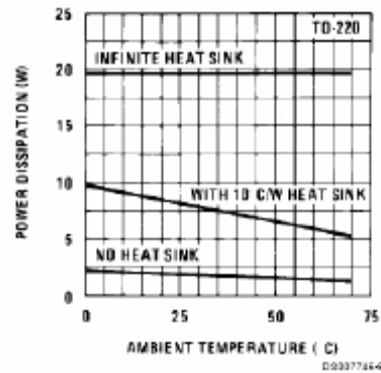
4.2.1 - Power Supply

The aim of this circuit is to regulate +5V and +9V from a 12V Sealed Lead Acid (SLA) battery or transformer source. Voltage to the regulator is controlled via a mechanical switch operated by the user, giving the circuit hardware on and off capability. The entire power circuit is fused at 2A in case of any power surges, and further protection is provided by the current protecting 1N1004 diode.

The voltage source to the circuit is from either a 12V SLA battery (default) or from a 12V transformer. Switching between sources is via a switched 2.5mm DC socket.

The +9V is generated through a LM7809 voltage regulator, to provide power to the minimal board. The other part of circuitry is to generate a regulated +5V power rail, using a LM7805 voltage regulator to provide power to the other devices such as all the sensors, servos and keypad decoder. The +5V regulator draws roughly 800mA and thus must have a heat sink to prevent overheating which could lead to malfunctioning of the device.

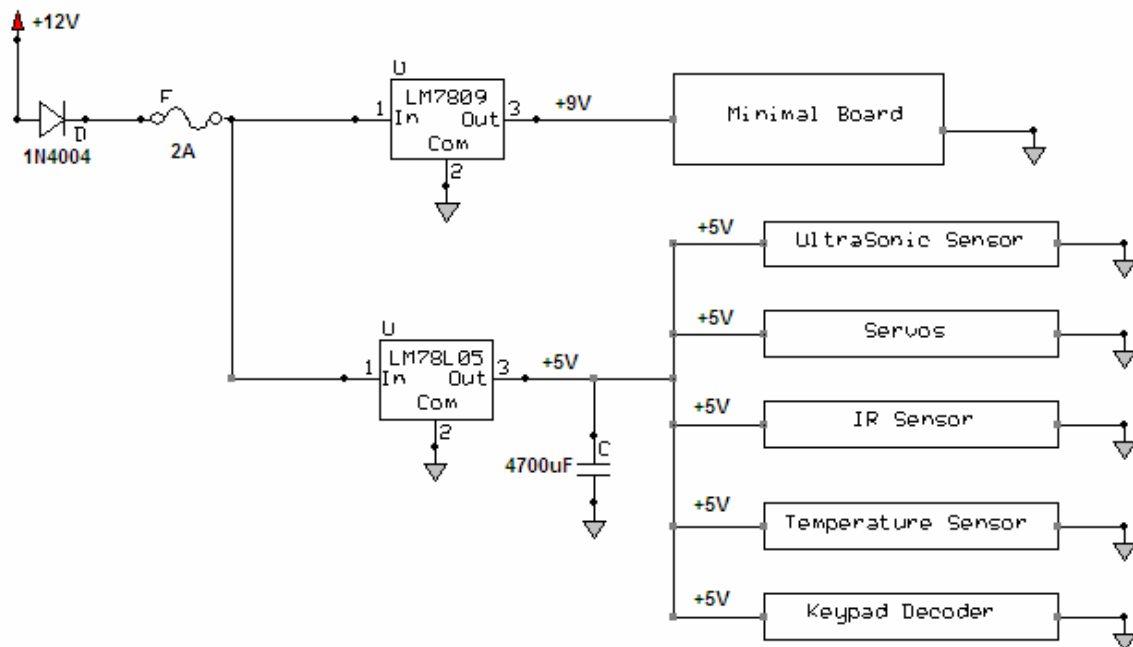
Maximum Average Power Dissipation



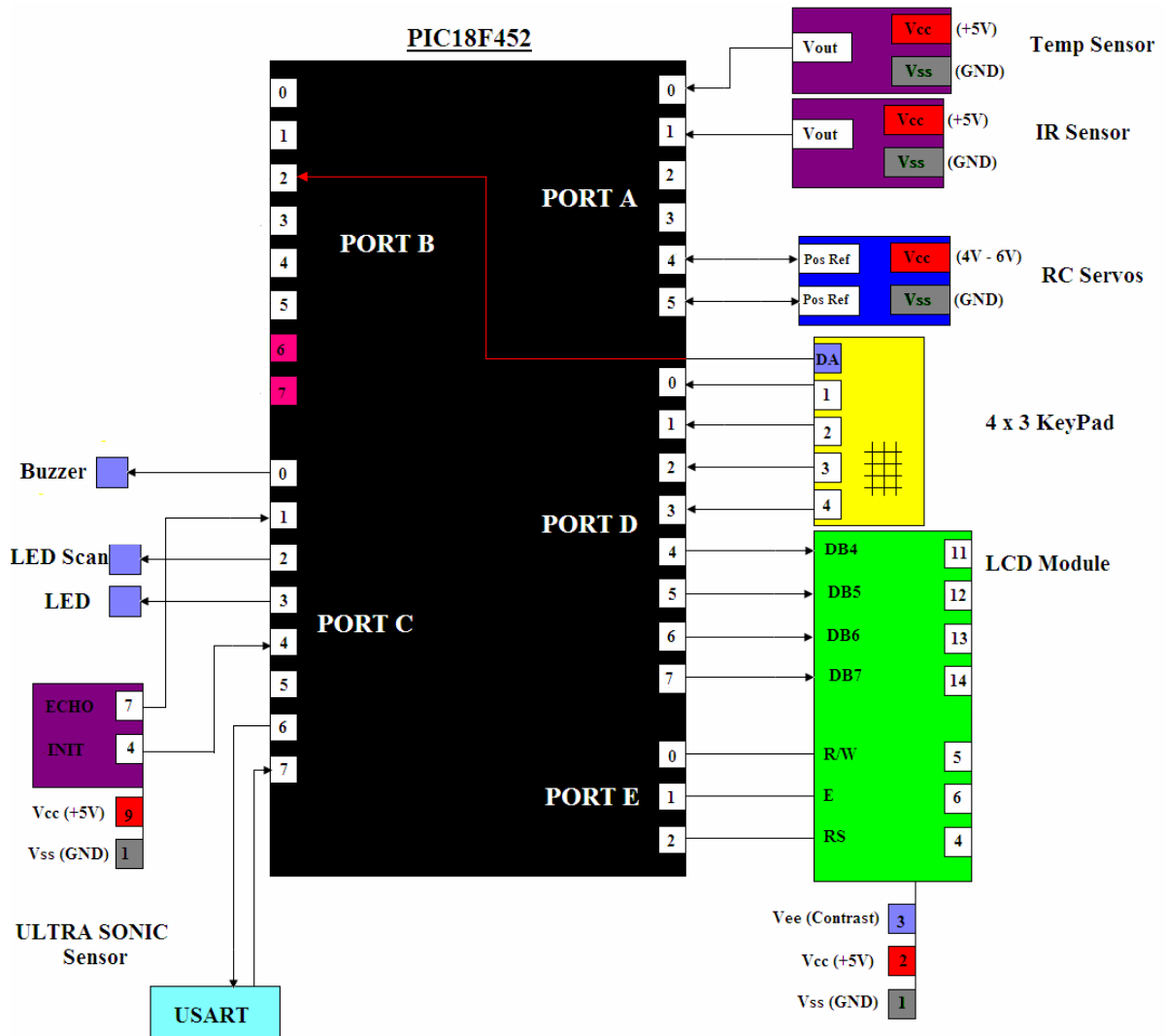
National Semiconductor LM78XX Series Voltage Regulators Datasheet, pp 5.

All devices are connected to a common ground such that there are no floating voltages. This enables the entire device to function efficiently and reduced the chance of any inconsistent behavior from the devices.

4.2.1.1 – Circuit Schematic



4.2.2 – Internal Connections (Pin Allocations)



All individual devices were grounded to a common ground rail and each device was run of a common +5V rail.

Every connection was crimped and soldered where possible. Headers and heat shrink were used to prevent open wires. The heat sinks were insulated from the wires to prevent any of the wires melting due to the heating up of the voltage regulators.

4.2.3 - Sensor Hardware

4.2.3.1 – US Sensor

The Ultra Sonic module operates of a +5V voltage source with the ground connection being commonly connected to the main ground of the system. The current draw from the Ultra sonic sensor is relatively high, 2A per impulse sent out; as such a capacitor was required to with stand the high current draw without running the battery flat.

The value of the capacitor C is worked out in accordance to the value current draw of the Ultra Sonic sensor. (See Appendix E)

The capacitance was calculated to be sufficient if over 4000uF and thus a 4700uF capacitor was used.

The only pin connections from the Ultra Sonic sensor that were used for the single-echo-mode was INIT and ECHO which were both connected to PORT C pin 4 and 1 respectively. As the INIT pin is sent high, the PIC would wait for a response on the echo pin before jumping into an interrupt to perform the distance and tracking algorithms.

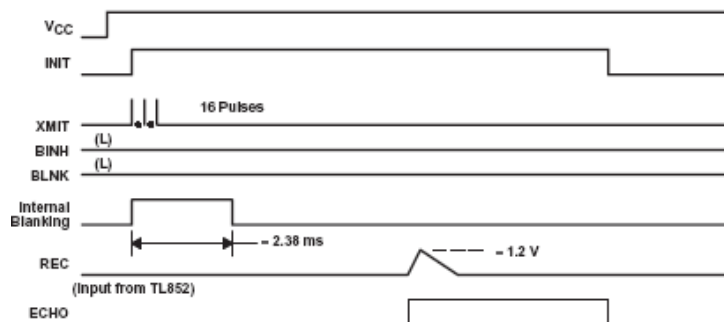
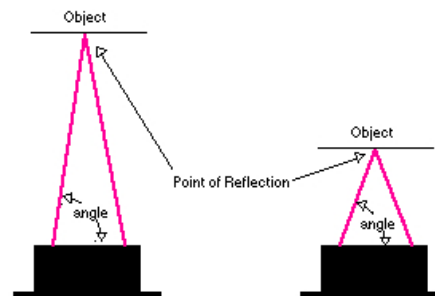


Figure 2. Example of Single-Echo-Mode Cycle When Used With the TL852 Receiver and 420-kHz Ceramic Resonator

4.2.3.2 - Sharp GP2Y0A02 Infrared Sensor Infra-Red Sensor

The operation of the IR sensor uses the triangulation and small linear CCD array to compute the distance of objects in the field of view. A pulse of IR light is emitted by the emitter which travels out in the field of view and either hits an object or just keeps going. In the case of no object, the light is never reflected and the reading shows no object. If the light reflects off an object, it returns to the detector and creates a triangle between the point of reflection, the emitter, and the detector (as shown below).



The angles in this triangle vary based on the distance to the object. The receiver portion of is a lens that transmits the reflected light onto various portions of the enclosed linear CCD array based on the angle of the triangle. The CCD array can then determine what angle the reflected light came back at and therefore, it can calculate the distance to the object within a range of 20cm to 150cm.

■ Absolute Maximum Ratings ($T_a=25^{\circ}\text{C}$)

Parameter	Symbol	Rating	Unit
Supply voltage	V_{CC}	-0.3 to +7	V
*1 Output terminal voltage	V_O	-0.3 to $V_{CC}+0.3$	V
Operating temperature	T_{opr}	-10 to +60	$^{\circ}\text{C}$
Storage temperature	T_{stg}	-40 to +70	$^{\circ}\text{C}$

*1 Open collector output

■ Recommended Operating Conditions

Parameter	Symbol	Rating	Unit
Operating Supply voltage	V_{CC}	4.5 to 5.5	V

From the specified operating conditions, the IR sensor was provided with +5V form the main 5V power rail and grounded to the common ground rail.

4.2.3.3 - National Semiconductor IC Temperature Sensor

The LM35 series are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Celsius (Centigrade) temperature. The LM35 thus has an advantage over linear temperature sensors calibrated in

$^{\circ}$ Kelvin, as the user is not required to subtract a large constant voltage from its output to obtain convenient Centigrade scaling.

The LM35 does not require any external calibration or trimming to provide typical accuracies of $\pm 1/4^{\circ}\text{C}$ at room temperature and $\pm 3/4^{\circ}\text{C}$ over a full -55 to $+150^{\circ}\text{C}$ temperature range.

Low cost is assured by trimming and calibration at the wafer level. The LM35's low output impedance, linear output, and precise inherent calibration make interfacing to readout or control circuitry especially easy. As it draws only $60\text{ }\mu\text{A}$ from its supply, it has very low self-heating, less than 0.1°C in still air. The LM35 is rated to operate over a -55° to $+150^{\circ}\text{C}$ temperature range.

Features:

- ❖ 0.5°C accuracy guarantee able (at $+25^{\circ}\text{C}$)
- ❖ Rated for full -55° to $+150^{\circ}\text{C}$ range
- ❖ Operates off 5V
- ❖ Less than $60\text{ }\mu\text{A}$ current drain
- ❖ Low self-heating, 0.08°C in still air
- ❖ Low impedance output, 0.1 W for 1 mA load
- ❖ Calibrated directly in $^{\circ}\text{C}$ (Centigrade)
- ❖ Linear $+10.0\text{ mV}/^{\circ}\text{C}$ scale factor

4.2.4 - Actuator Hardware

4.21.4.1 - Servos

The RC Servos used for the device were the JR model NES-537. This is a standard sized analogue servo with output shaft and internal gears made out of plastic. They servos operate under the basic specifications listed below:

Table 1: Basic Specification - JR Model NES-537

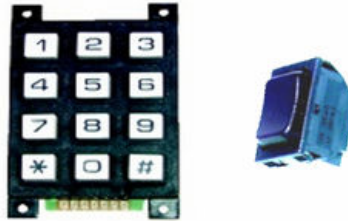
Quantity	Value
Dimensions	32.5 x 19.0 x 38.5 mm
Weight	45 g
Speed	$260^{\circ}/\text{sec}$
Voltage	4.8 V DC
Torque	0.32 N.m

Table 2: JR Servo Pinout

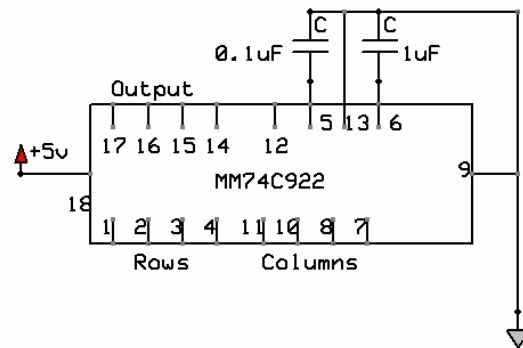
Function	Colour
+0.0 V DC	Brown
+4.8 V DC	Red
Position Ref	Orange

4.2.5 - Operator Input Hardware

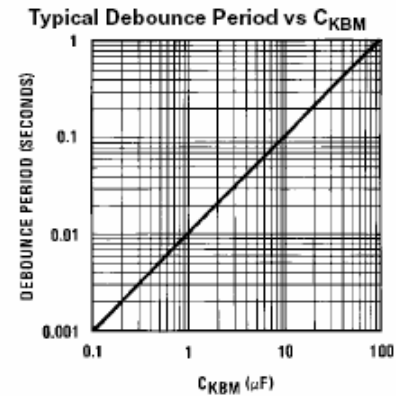
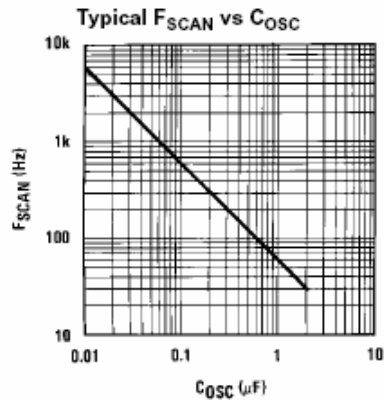
4.2.5.1 - Keypad



The 4x3 keypad works using the principle where the 4 rows and 3 columns are railed independently such that when each row is provided with 5V and the columns are railed at 0V. When a button is pressed, the output on the decoder chip (MM74C922) is independent to the output provided from a second button. Each button has its individual 4 bit expression that is driven to PORT D pins 0 to 4. This enables the PIC to read off exactly which button is being pressed.



Each individual row is connected to pins 1 to 4 of the decoder chip and each column is connected to pin 7, 8 and 10. The common pin on the rocker switch is connected to pin 11, while the other two pins of the switch is connected in parallel to pins 1 and 2. The circuitry for the keypad decoder requires two capacitors to control the debounce of the switches. In accordance to the MM74C922 datasheet, the value of the capacitor at the Keybounce Mask pin should be 10 times the value of the OSC pin. Following the data of the following graphs, the value of the capacitors was chosen to be 0.1uF and 1uF. These values enabled a relatively smaller bounce period of 0.01 seconds and a faster scan period of 1 KHz. After considerable testing, these values were found to be suitable for our application.



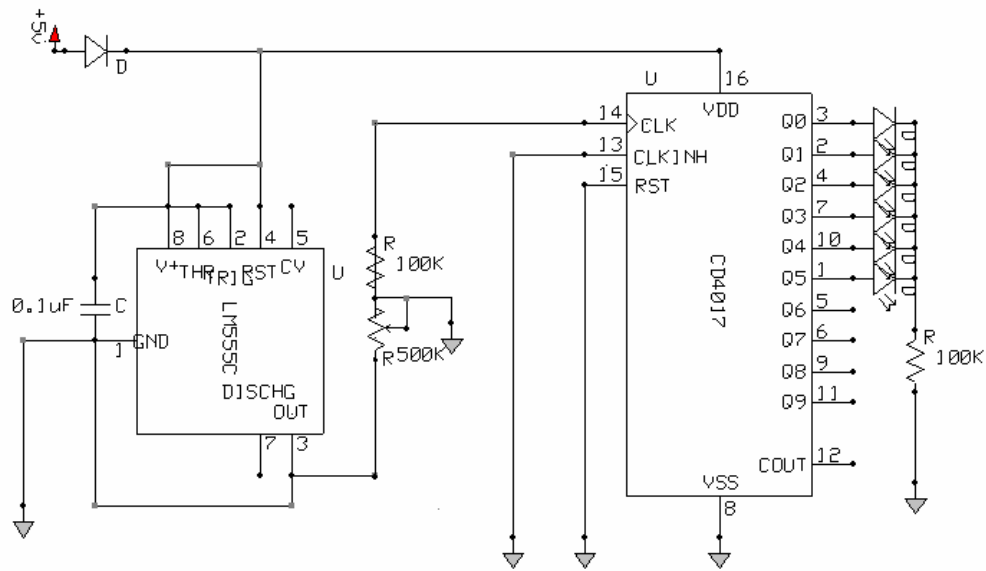
The table in appendix F shows the significant value of each key to its 4 bit number.

4.2.6 - Operator Output Hardware

4.2.6.1 - LED Scanning & Tracking

The idea behind the LED's for scan and tracking was to provide a visual display for the user to know whether the DST is in scanning or tracking mode. The tracking LED (green) is a flashing LED which is driven by PORT C pin 4 when the device is in tracking mode. The LED scan module is driven by the following circuit. A CMOS 4017 IC decade counter is used to continuously output an LED on. The clock of this circuit is driven by a LM555 timer. The clock output can be adjusted using a variable resistor such that the rate of scan can be set by the user. PORT C pin 3 was set as an output pin and driven high when the device is in scanning mode.

The use of the CMOS 4017 decade counter required sensitive use due to the static nature of the IC. Newer models of decade counters were considered, however, the ease of use and previous use of the 4017 chip enable an efficient circuit to be designed.



4.2.6.2 - LCD

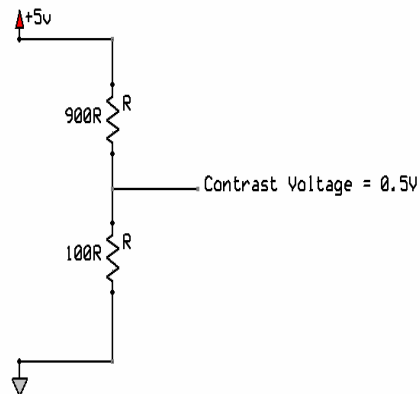
The LCD module used by DST was a large LCD module. The large character size enables the user to view the LCD from further distances, easier on the eyes of the user and is mounted behind Perspex for protection. Intricate specifications of the LCD include: 16 characters x 2 lines, 5V operating voltage, 96 built in ASCII characters, 92-special characters and 8 custom characters. With its own backlight, the LCD module provides an attractive appeal to the user and enhances the quality of the hand controller.

The pin allocation of the LCD follows the table below. However, the LCD module was used in 4-bit mode to enable pins to be free on the PIC. Using 8-bit mode would enable a faster application, however it was decided that the 4-bit mode was sufficient enough for the use as the user can only read at a generic rate. Thus there was no advantage of using 8-bit more.

<u>Pin Assignments</u>		
Pin No.	Label	Description
1	Vss	Ground
2	VDD	Supply Voltage
3	VO	Contrast adjustment voltage
4	RS	Register select signal
5	R/W	Read / write select signal
6	E	Operation (read/write) enable
7	DB0	Low byte data bit
8	DB1	Low byte data bit
9	DB2	Low byte data bit
10	DB3	Low byte data bit
11	DB4	High byte data bit
12	DB5	High byte data bit
13	DB6	High byte data bit
14	DB7	High byte data bit
15	A	Positive LED backlight (Anode)*
16	K	Negative LED backlight (Cathode)*

* Backlighting connections for Z 7011 Only

The RS, R/W and E pins of the LCD were connected to the PIC as per the pin allocation diagram previously displayed. The supply of the contrast voltage is shown below. A voltage divider was used to produce an output of 0.5V to the VO pin. This contrast voltage could have been changed with a potentiometer being added, however it was decided that the user should not have control of the contrast to the LCD as more hardware components could eventually lead to problematic issues.



4.2.7 - Hardware Quality Assurance

- ❖ Circuits were continuity tested to ensure the construction was free of solder bridges and components were correctly placed.
- ❖ The system was powered at full power consumption and voltage regulator temperature measured to ensure safe temperature levels.
- ❖ Power supply contains reverse polarity protection diodes and fuses to protect the device from large voltage spikes.
- ❖ Individual boards were powered from a single rail in star point configuration to reduce supply voltage problems such as bounce, ground loops etc.

4.3 - Hardware Validation

Throughout this entire project, the hardware had to be continually tested and checked to make sure that the system was operating to the correct safety and functionality standards. They included

- ❖ Using a multimeter to check all connections
- ❖ Making sure all wires had crimps, headers and heat shrinks
- ❖ Making sure the heat sinks were working to keep the system safe

Once all these steps had been taken for a particular element of this system, it was secured and protected from further damage. Eg: once the keypad, rocker switch and LCD were verified to be working correctly, they were secured to the User Interface enclosure using electrical tape, hot glue and screws so that they could not be accidentally damaged.

4.4 - Hardware Calibration Procedures

The calibration of the LED Scan circuit can be controlled by the user within the hand held controller. A variable resistor that controls the speed of scan of the LED's has easy access for the user. The calibration can be controlled between 100K and 500K to the user's deliberation.

Further calibration can be performed through software modules in factory mode.

4.5 - Hardware Maintenance and Adjustment

For all the hardware that this project comprised of, no specific maintenance was required; only the issue of safe storage and transportation. Each component had to be safely handled under all circumstances to avoid damage which would ultimately lead to system failure.

Components had to be also protected from dust and external bugs that could potentially interfere with system functionality.

Holes were also drilled into the Sensor Array enclosure to provide for the heat build up inside the box due to the heatsinks' operation.

5. SOFTWARE DESIGN

The software requirements and overview have been dealt with elsewhere in this document. The present section addresses the design and implementation of the software that forms the Death-Star-Tracking system.

5.1 - Software Design Process

The software was designed in a modular fashion, with a bottom-up approach. Firstly, the entire operation of the system was investigated and decided upon. This involved the creation of a state-flow-diagram, and from this the project was separated into a number of modules that were assigned to Team Cinder technicians. Following this, each module was further broken down into the constituent functionalities/requirements and these were designed and created in order to attain module functionality. Following the completion of functioning modules, integration was done between them to eventually to create a singular program.

5.2 - Software Development Environment

The software development environment involved the use of MPLAB on a PC. This piece of software provided the appropriate environment for compilation and assembly of C code to translate into machine code for use by the microprocessor. The hardware involved in the software development environment included a set of probes, a digital oscilloscope, a digital voltmeter and breadboard circuitry. 2 different PIC18f452 microcontroller arrangements were also used for the development of software, those being the development board and the minimal board.

5.3 -Software Implementation Stages and Test Plans

Implementation stages followed a simple progression:

- ❖ Understand the module requirements
- ❖ Make the hardware work using purely module-specific functional code and understand its operation
- ❖ Create code appropriate to the specific requirements of the module
- ❖ Test
- ❖ Integrate with other modules
- ❖ Test

Each module was designed to have as many “private” functions as possible, with only inter-module-reliant variables and functions given the global designation. This enabled the functionality of different modules to be insulated from others, with the idea being that bugs in code would be easier to diagnose and fix. Inter-reliant aspects of the system were identified during the system definition and state-flow diagram creation.

Design within modules was first done by investigating the appropriate data sheets and gaining a succinct understanding of how the module was to achieve its

objectives. This sometimes required writing code to get hardware working, but not in a form to suit that of the module requirements. An example of this was initially getting the IR hardware to function and understand its output by writing an individual function that polled the A/D conversion complete bit. This allowed the Team Cinder technician to gain a very good understanding of the operation of the IR hardware and its expected output. Following this, new code was written for the IR sensor that involved interrupts that would suit the operation of the module far better.

Another code design tool used was writing pseudo code. This was often done during the process of understanding the requirements of the module. A series of dot points were generated from understanding the requirements of the module. For example:

- ❖ Interrupt Driven – check and change flags only in interrupt routine
- ❖ Main mathematical function running in endless while loop
- ❖ Only change key display variables once all calculations are complete.

From these dot points, pseudo code was written in the same structure that the code would need to follow. An example from IR implementation follows:

Setup the appropriate hardware bits for input/output – PORTA pin 1 = input
Setup the AD converter and interrupt characteristics

```
while(1)
{
    Check IR_calculate flag
    Take the value of the AD_conversion
    Calculate the distance
    Update 'Distance' Variable
}

Interrupt routine
{
    Check what triggered interrupt
    If AD con done triggered, then set 'IR_calculate' flag
}
```

Following this, the actual functional code would be written.

Testing procedures involved both hardware and software utilisation. For example, much code was tested by setting breakpoints in MPLAB, which would identify when, if ever, certain parts of code were executed by the microprocessor. Also, MPLAB was utilised to step through code step by step, using the watch window to keep a track of changes in variable values. Another software method of testing involved the stopwatch feature in the MPLAB debugger in simulation mode. Also, the use of probes, the digital oscilloscope and a digital voltmeter were used for testing input and output states of different pins. Integration testing involved the use of these aforementioned devices also.

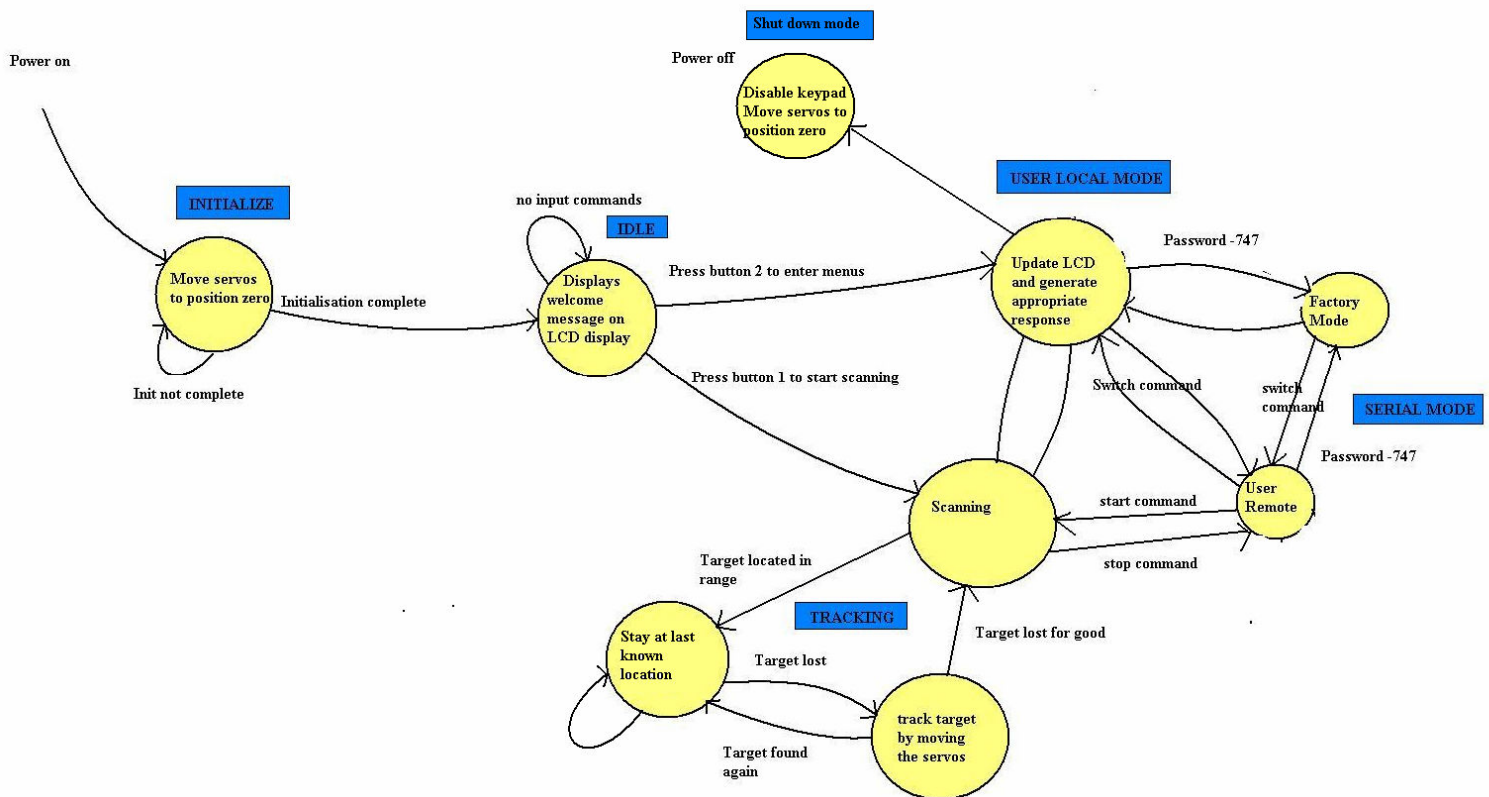
5.4 - Software Quality Assurance

Software quality was controlled using a number of methods. Firstly, all mathematical functions were created in a way to minimise the time taken perform the operations. This included applying bit-shifting instead of multiplication or division wherever possible, and avoiding floating point arithmetic. Secondly, all variables that were module-specific were defined in a header file for that module. This made the code easier to read. Thirdly, there was a conscious effort to eliminate any need for 'magic numbers' by defining such variables with meaningful names and values in the appropriate header file.

5.5 - Architecture

5.5.1 - State transition diagram

Team Cinder death star tracker. State transition diagram



The software running our death star tracker is entirely interrupt driven except when an update is required for the LCD. As shown on the state transition diagram the program starts up by initializing the different modules and displaying a welcome message on the LCD. As stated in the User manual, the User is then given the choice to either start scanning or navigate through the menus. At any time the user can change the different settings available to him, start and stop scanning, or switch to one of the two modes available via the serial connection. Factory mode is password protected to prevent the User from accidentally changing calibration settings and as shown on the diagram, the user is able to navigate the between the 3 modes at will.

5.5.2 - PWM module.

The two servos of the device are controlled by two different and independent PWM generated from one single Timer interrupt Timer 1.

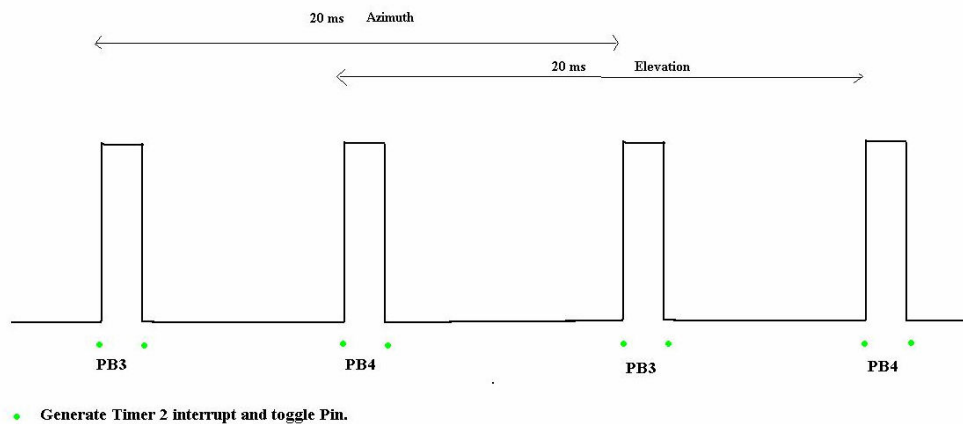


Figure 2. PWM module

The servo's are by far the most important part of the device and should have priority over any other modules. Indeed the duty cycle of the signal being sent must be between 1ms and 2ms. We did not want anything to distract the toggling of the pins when the timer1 interrupt would fire and thus set timer 1 to high priority, while all other interrupts within the program are set to low priority.

The value in CCP2 is constantly changed to get Timer 1 to fire in a coordinated fashion to obtain 2 PWM's independent of each other. These values are recalculated when necessary by calling the function calculation().

5.5.3 - Interrupt Handlers

Because there are only 16 bytes of available memory location between the two interrupts vector addresses (0x0008 and 0x0018), the program counter is loaded with the address of an interrupt_handler sub-routine. As shown on the graph here below the low priority interrupt handler, handles all of the program's interrupts except for timer 2.

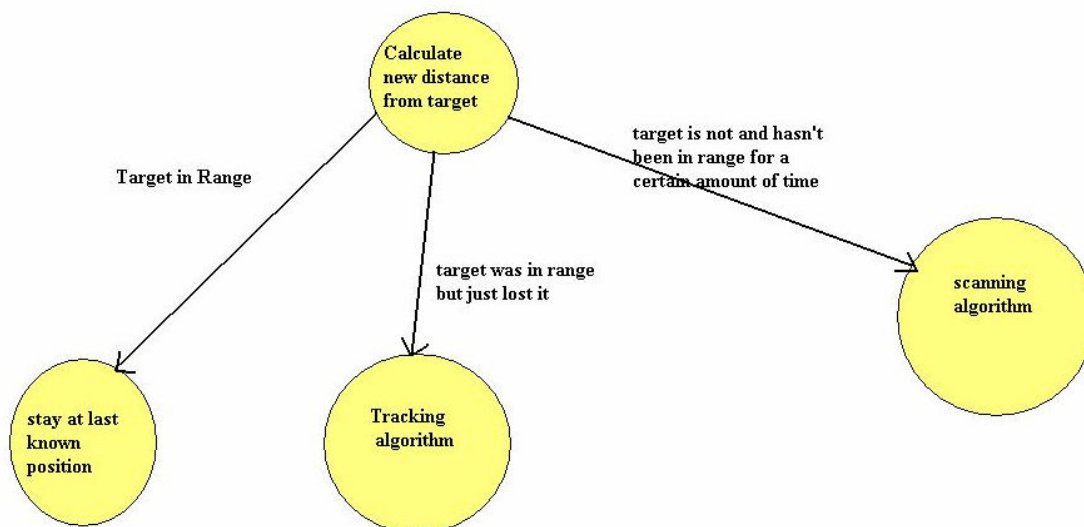
<p>Interrupt Handler Low</p> <p>TIMER 0 Timer 0 overflow interrupt Used to send an ultrasonic pulse or and IR signal</p> <p>CCP2IF capture compare 2 interrupt Used to capture the echo from the ultra-sonic sensor.</p> <p>ADIF A/D conversion interrupt Used for Infra-red and temperature sensors, in order to know when the A/D conversion has finished.</p> <p>TXIF serial transmit interrup Used in transmitint characters through serial</p> <p>RCIF Serial receive interrupt Used in receiving characters through serial</p> <p>INT2IF Rise edge interrupt Used in User_local mode, in receiving characters from the keypad</p> <p>TIMER 2 Timer 2 interrupt Used to generate a bip with the buzzer.</p>	
--	--

Low priority interrupt handler

As a result of the nature of the program being almost entirely interrupt driven, the program executes in an infinite loop and waits for the different interrupts to trigger in order to generate the appropriate response.

5.5.4 - Tracking and scanning transition.

An important part of the program is the transition between scanning and tracking. During either one of these two stages, the next position of the servos is recalculated, only when a new echo from the Ultra-sonic sensor has been detected. When that happens the average distance from the device to its target is calculated, and only then is the calculation function called to move the servos to the next desired location. This implies that the frequency at which the ultra-sonic sensor is being fired has an effect on the rotational speed of the servos during scanning and tracking.



Tracking state transition

5.6 / 5.7 - Software Interface / Software Components

Each of the software modules are described in detail from section 2 and earlier sections of 5. Their conceptual design, internal workings, implementation and public interfaces were all overlapped in their specific sections to provide the reader with a detailed description on their behaviour.

5.8 - Preconditions for System Start-up

System start-up was triggered first using hardware which would subsequently initialize the software modules.

After the user had turned to the system on, all the initialisation function would run and the LCD, Keypad, Buttons, User Interface, Sensors and LED's would be ready for operation.

System start-up can be verified by the LCD screen which will display a welcome message as shown in section3, and the servos which will move to the calibrated [0 0] position ready for user input.

Before the system can actually be started, the power must be connected to the entire system and the PIC board must be correctly connected. This will already be done when the final product is ready for testing / sale.

5.9 - Preconditions for System Shutdown

As per system start-up, shutdown will occur in the reverse order.

When the User triggers the proper4 system shutdown sequence by selecting "Shutdown" from the options menu of User Local mode, the Servos will move to the calibrated [0 0] position.

All hardware will be de-initialized through software and the LCD, LED's, Keypad, Buttons and extra components will be turned off.

The only way for the user to enter the system again for usage is to turn the entire system off and on using the power rocker switch.

6. SYSTEM PERFORMANCE

6.1 - Performance Testing

The testing was broken down into 3 sub categories;

6.1.1 - IR and US Sensor

We used both a tape measure and the watch window feature of MPLAB to correct and account for distance irregularities during the sensors' calibration.

6.1.2 – Temperature sensor

The temperature sensor was placed outside the enclosure to measure the exact air conditions and then factor the value into calculations. We used a thermometer (located near the sensor to provide a reliable comparison) to measure the temperature at any time period required and could thus offset our temperature calculation value accordingly.

6.1.3 – Memory Usage

By constantly viewing the memory available to us by the PIC board using MPLAB, we were aware at all times from where the program was actually running and could thus transfer memory from Program → Data and vice versa. We found that the final integrated file with all components used up less memory than each of the separate components, mostly due to the usage of multiple while(1) loops.

6.2 - State of the System as Delivered

In the opinion of group (Team Cinder), this product operated to roughly 90% of its required specification. The only things missing in our final product were:

4 Factory Mode settings (the user could still access the settings but the function for the setting itself had not been written)

Random system crash – Sometimes the Keypad and LCD module would interfere in such a way that the system would stop functioning and an entire restart was required.

Besides these 2 minor components, all settings and modes of operation plus extra features including;

- ❖ Battery – power and recharge capabilities
- ❖ Buzzer – could be turned on and off at users discretion
- ❖ The ability to enter 2D tracking from 3D tracking mode
- ❖ LED display

- ❖ Big LCD with backlight
- ❖ Safe and ergonomic and aesthetically pleasing final product
- ❖ Power and Charging Light

functioned to specifications and met the needs required by the MTRX3700 assignment outline.

6.3 - Future Improvements

Because our current DST incorporated so many user friendly extras, not many more improvements need to be made. Besides the addition of the missing factory mode settings, a few factors that could be improved upon were;

1. Improvement was concerning the layout of the components inside the Sensor Array Enclosure. Although from the exterior it was sleek and well designed, inside was basically a jungle of wires that could possibly disrupt the system during operation. In Future releases of the DST, a bigger enclosure could be used to house all the components and make it easier in terms of testing and repair.
2. Labels on the external buttons: If the user had lost the user manual then using the system for the first time would be extremely hard. Although having labels on the enclosures exterior would wreck its appearance, it would make the users job of understanding and working their way a round the system for the first time easier.

7. SAFETY IMPLICATIONS

The DST comprises of numerous electrical connections and components that could be a potential health risk to all users.

7.1 – Operational Conditions

- ❖ The voltage regulator can reach elevated temperatures up to 80°C and is a burning hazard. Please be observant of this when touching internal circuitry, specifically the voltage regulator and heat sink.
- ❖ The system is not extensively tested and there is a possible fire risk from overheating. For this reason the system should not be left powered on without supervision. A serious transformer failure could produce 250V AC at the DC plug and present an electrocution hazard. This however is very unlikely.
- ❖ Possible shorts in the system due to exposed conduction surfaces. No loose wires are left free in the system that might create shorts. Any unused wires we eliminated and heat shrink tubes,

7.2. User's Perspective

- ❖ Water And Moisture – The appliance should not be used near water – for example, near a bathtub, washbowl, kitchen sink, laundry tub, in a wet basement, or near a swimming pool and the like. To help prevent moisture damage and shorts, all cables and components that make up this product are completely are sealed and enclosed. • Cleaning – The appliance should only be cleaned with a moist towel or sponge with care taken not to introduce moisture into the enclosure or any external sockets.

All electrical connections are housed within these 3 components and their premise must NEVER be breached.

Team Cinder highly recommends that the user DOES NOT try and modify this product in any way/shape or form without consulting a qualified technician.

It is not a toy and should be handled with utmost care AT ALL TIMES.

8. CONCLUSIONS

The Death Star Tracker is an innovative product which utilises both an Ultra Sonic and Infra-Red sensor to scan, track and find an object to give an accurate reading of its position and distance in a specified region of free space. The system is low power, lightweight, portable; and can be operated by a user of any technical level.

The DST is basically a small and inept model of most modern tracking systems which play a major part in our society today with applications ranging from Hawkeye technology used in cricket matches to complex space-bound tracking systems.

By studying the specifics of this system, the user gains a thorough understanding on the nature and workings of products such as this; and can in the future, apply this knowledge to design and create the very things that encapsulate human evolution.

APPENDIX A:

Updating the 'IR_distance' value

$$Volts = \frac{5 * IR_AD_result}{1023}$$

$$IR_distance = \frac{9400 * Volts}{17 * Volts^2 - 2}$$

APPENDIX B:

Menu Number	Menu
0	Start-up Menu
1	Show Target Status
2	Show Temperature
3	Set Minimum Azimuth
4	Set Maximum Azimuth
5	Set Minimum Elevation
6	Set Maximum Elevation
7	Set Minimum Range
8	Set Maximum Range
9	Goto Azimuth
10	Goto Elevation
11	Set Factory Mode
12	Set User Remote Mode
13	Exit Menu
14	Scanning/Tracking Mode Display
15	Remote/Factory Mode Display
16	View Min/Max Azimuth
17	View Min/Max Elevation
18	View Min/Max Range
19	Stop/Start Scanning/Tracking
20	Options Menu
21	Shutdown Menu

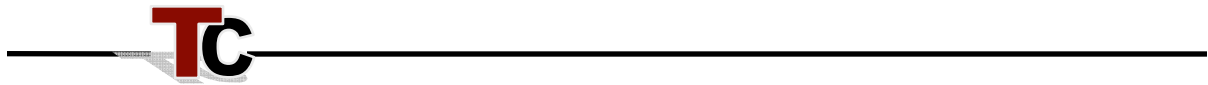
0	Blank String	24	Confirm Change
1	Welcome To Team	25	User Remote
2	Cinder DST	26	Exit Menu
3	To Scan Press 1	27	
4	For Menu Press 2	28	Tracking...
5	Show The Current	29	Scanning...
6	Target Status	30	Menu: Any Key
7	(xxx,xxx,xxx)	31	Current Mode:
8	Temperature	32	Not Active...
9	xx.x Degrees C	33	Display Min/Max
10	Set Minimum	34	Min: xxxx
11	Set Maximum	35	Max: xxxx
12	Range	36	Stop
13	Elevation	37	Start
14	Azimuth	38	Options Menu
15	xxxx	39	1: Enable
16	1:Yes	40	1: Disable
17	Incorrect Entry	41	3D-Tracking
18	1:Retry	42	Buzzer
19	Go To	43	Shutdown
20	Change Mode To	44	Unplug Serial
21	Factory	45	Cable & Press #
22	Enter Code	46	DST Is Now Safe
23	xxxx	47	To Turn Off

APPENDIX C:

Menu Level	Menu Depth	LCD Line	String #	String	Alternate Strings	
0	0	0	1	Welcome To Team		
0	0	1	2	Cinder DST		
0	1	0	3	To Scan Press 1		
0	1	1	4	For Menu Press 2		
1	0	0	5	Show The Current		
1	0	1	6	Target Status		
1	1	0	6	Target Status		
1	1	1	7	(xxx,xxx,xxxx)	32 29	Not Active... Scanning...
2	0	0	5	Show The Current		
2	0	1	8	Temperature		
2	1	0	8	Temperature		
2	1	1	9	xx.x Degrees C		
3	0	0	10	Set Minimum		
3	0	1	14	Azimuth		
3	1	0	14	Azimuth		
3	1	1	15	xxxx		
3	2	0	15	xxxx		
3	2	1	16	1:Yes		
3	3	0	17	Incorrect Entry		
3	3	1	18	1:Retry		
4	0	0	10	Set Maximum		
4	0	1	14	Azimuth		
4	1	0	14	Azimuth		
4	1	1	15	xxxx		
4	2	0	15	xxxx		
4	2	1	16	1:Yes		
4	3	0	17	Incorrect Entry		
4	3	1	18	1:Retry		
5	0	0	10	Set Minimum		
5	0	1	13	Elevation		
5	1	0	13	Elevation		
5	1	1	15	xxxx		
5	2	0	15	xxxx		
5	2	1	16	1:Yes		
5	3	0	17	Incorrect Entry		

5	3	1	18	1:Retry		
6	0	0	11	Set Maximum		
6	0	1	13	Elevation		
6	1	0	13	Elevation		
6	1	1	15	xxxx		
6	2	0	15	xxxx		
6	2	1	16	1:Yes		
6	3	0	17	Incorrect Entry		
6	3	1	18	1:Retry		
7	0	0	10	Set Minimum		
7	0	1	12	Range		
7	1	0	12	Range		
7	1	1	15	xxxx		
7	2	0	15	xxxx		
7	2	1	16	1:Yes		
7	3	0	17	Incorrect Entry		
7	3	1	18	1:Retry		
8	0	0	11	Set Maximum		
8	0	1	12	Range		
8	1	0	12	Range		
8	1	1	15	xxxx		
8	2	0	15	xxxx		
8	2	1	16	1:Yes		
8	3	0	17	Incorrect Entry		
8	3	1	18	1:Retry		
9	0	0	19	Go To		
9	0	1	14	Azimuth		
9	1	0	14	Azimuth		
9	1	1	15	xxxx		
9	2	0	15	xxxx		
9	2	1	16	1:Yes		
9	3	0	17	Incorrect Entry		
9	3	1	18	1:Retry		
10	0	0	19	Go To		
10	0	1	13	Elevation		
10	1	0	13	Elevation		
10	1	1	15	xxxx		
10	2	0	15	xxxx		
10	2	1	16	1:Yes		
10	3	0	17	Incorrect Entry		
10	3	1	18	1:Retry		
11	0	0	20	Change Mode To		
11	0	1	21	Factory		
11	1	0	22	Enter Code		
11	1	1	23	xxxx		
11	2	0	24	Confirm Change		

11	2	1	16	1:Yes		
11	3	0	17	Incorrect Entry		
11	3	1	18	1:Retry		
12	0	0	20	Change Mode To		
12	0	1	25	User Remote		
12	1	0	24	Confirm Change		
12	1	1	16	1:Yes		
13	0	0	26	Exit Menu		
13	0	1	27			
13	1	0	24	Confirm Change		
13	1	1	16	1:Yes		
14	0	0	29	Scanning...		
14	0	1	30	Menu: Any Key		
14	1	0	28	Tracking...		
14	1	1	30	Menu: Any Key		
15	0	0	31	Current Mode:		
15	0	1	21	Factory		
15	1	0	31	Current Mode:		
15	1	1	25	User Remote		
16	0	0	33	Display Min/Max		
16	0	1	14	Azimuth		
16	1	0	34	Min: xxxx		
16	1	1	35	Max: xxxx		
17	0	0	33	Display Min/Max		
17	0	1	13	Elevation		
17	1	0	34	Min: xxxx		
17	1	1	35	Max: xxxx		
18	0	0	33	Display Min/Max		
18	0	1	12	Range		
18	1	0	34	Min: xxxx		
18	1	1	35	Max: xxxx		
19	0	0	36	Stop	37	Start
19	0	1	29	Scanning...	28	Tracking...
19	1	0	24	Confirm Change		
19	1	1	16	1:Yes		
20	0	0	38	Options Menu		
20	0	1	27			
20	1	0	39	1: Enable	40	1: Disable
20	1	1	41	3D-Tracking		
20	2	0	39	1: Enable	40	1: Disable
20	2	1	42	Buzzer		
20	3	0	39	1: Enable		
20	3	1	43	Shutdown		
21	0	0	44	Unplug Serial		
21	0	1	45	Cable & Press #		
21	1	0	46	DST Is Now Safe		



21	1	1	47	To Turn Off		
----	---	---	----	-------------	--	--

APPENDIX D:

Initial write-up of the lookup table

0,0,0	1	13 0 0	26	14 0 0	29
0,0,1	2	13 0 1	27	14 0 1	30
0,1,0	3	13 1 0	24	14 1 0	28
0,1,1	4	13 1 1	16	14 1 1	20

For Menu mode (=1):

(Menu, Depth, Line)	String		
5+d+L	str		
1,0,0	5	5 2 0	15
1,0,1	6	5 2 1	16
1,1,0	6	5 3 0	17
1,1,1	7	5 3 1	18
2,0,0	5	6 0 0	11
2,0,1	8	6 0 1	13
2,1,0	8	6 1 0	13
2,1,1	9	6 1 1	15
3,0,0	10	6 2 0	15
3,0,1	14	6 2 1	16
3,1,0	14	6 3 0	17
3,1,1	15	6 3 1	18
3,2,0	15	7 0 0	10
3,2,1	16	7 0 1	12
3,3,0	17	7 1 0	12
3,3,1	18	7 1 1	15
4,0,0	11	7 2 0	15
4,0,1	14	7 2 1	16
4,1,0	14	7 3 0	17
4,1,1	15	7 3 1	18
4,2,0	15	8 0 0	11
4,2,1	16	8 0 1	12
4,3,0	17	8 1 0	12
4,3,1	18	8 1 1	15
5,0,0	10	8 2 0	15
5,0,1	13	8 2 1	16
5,1,0	13	8 3 0	17
5,1,1	15	8 3 1	18

15 0 0	31
15 0 1	21
15 1 0	31
15 1 1	25
9 0 0	19
9 0 1	14
9 1 0	14
9 1 1	15
9 2 0	15
9 2 1	16
9 3 0	17
9 3 1	18
10 0 0	19
10 0 1	13
10 1 0	13
10 1 1	15
10 2 0	15
10 2 1	16
10 3 0	17
10 3 1	18
11 0 0	20
11 0 1	21
11 1 0	22
11 1 1	23
11 2 0	24
11 2 1	16
11 3 0	17
11 3 1	18
12 0 0	20
12 0 1	25
12 1 0	24
12 1 1	16

Initial string definition. Note how string 23 and string 15 were initially different to indicate their different usages.

(menu, track, menu, depth)
mode

Start up: 1 "Welcome to team" (0, 0, 0, 0)
2 "cinder DST" (0, 0, 0, 1)
3 "To scan press x" (0, 0, 0, 1)
4 "For menu press y" (0, 0, 0, 1)
Show_target_status 5 "Show the target" (0, 0, 1, 0)
6 "status Target status" (0, 0, 1, 0)
Updateable!! 7 "Target status:" (1, 0, 1, 1)
8 "(xxx, xxx, xxx)" (1, 0, 1, 1)
Show-temp 5 "Show current" (1, 0, 1, 2, 0)
8 "Temperature" (1, 0, 1, 2, 0)
8 "Temperature" (1, 0, 1, 2, 1)
9 "xxx deg C" (1, 0, 1, 2, 1)
Set_Azimuth-Min
range
elevation
10 "Set Min" (1, 0, 1, 3-8, 0-3)
11 "Set Max" (1, 0, 1, 3-8, 0-3)
12 "Range" (1, 0, 1, 3-8, 0-3)
13 "Elevation" (1, 0, 1, 3-8, 0-3)
14 "Azimuth" (1, 0, 1, 3-8, 0-3)
15 "xxx" (1, 0, 1, 3-8, 0-3)
Changeable Link
strings
16 "1: Yes 2: No" (1, 0, 1, 3-8, 0-3)
17 "Insert Entry" (1, 0, 1, 3-8, 0-3)
18 "Please try again" (1, 0, 1, 3-8, 0-3)
19 "Goto" (1, 0, 1, 3-8, 0-3)
Goto-Az 9
el 10
Set-factory
20 "Change mode to" (1, 0, 1, 3-8, 0-3)
21 "factory" (1, 0, 1, 3-8, 0-3)
22 "Enter code" (1, 0, 1, 3-8, 0-3)
23 "xxx" (1, 0, 1, 3-8, 0-3)
24 "Confirm Change" (1, 0, 1, 3-8, 0-3)
25 "User Reprote" (1, 0, 1, 3-8, 0-3)
26 "Exit Menu" (1, 0, 1, 3-8, 0-3)
27 "Blank space" (1, 0, 1, 3-8, 0-3)
28 "Tracking..." (1, 0, 1, 3-8, 0-3)
29 "Scanning..." (1, 0, 1, 3-8, 0-3)
30 "Main: Am Key" (1, 0, 1, 3-8, 0-3)
31 "Current Mode" (1, 0, 1, 3-8, 0-3)

Initial mathematics for the string definitions. Note the use of bit-shifts wherever possible to reduce computation time.

211 = 9 if ^{trading} scanning 32 if neither.
 = 29 if scanning.

Case (11) \leq put into
 if (d < 1)
 temp = 20 + d << 1 + L;
 if (d == 2, L == 0)
 temp = 24;
 else
 temp = 11 + d << 1 + L;
 $9 - (1-d)^2$
 $9 - (1-d)^2(1-L)^2$
 $9 - (2-d-L)^2$

temp.
 Switch (menu).
 case (1)
 temp = 5 + d + L;
 case (2)
 temp = 9 - (2-d-L) * (2-d-L)
 case (3 || 4 || 5 || 6 || 7 || 8)
 if (d > 2) {
 temp = 11 + d << 1 + L;
 }
 else if (d == 1 && L == 1)
 temp = 15;
 else if (d == 0 && L == 0)
 temp = 11 - ((menu >> 1) << 1) + menu;
 else {
 temp = 19 - menu;
 }
 case (9 || 10)
 if (d > 2)
 temp = 11 + d << 1 + L;
 if (d == L)
 temp = 19 - d << 1 - L << 1;
 else
 temp = 23 - menu;

APPENDIX E:

The value of the capacitor C

$$I_c = C \frac{dV_{in}}{dt}$$

$$C = I_c \frac{\Delta t}{\Delta V}$$

$$C = 2A \times \frac{1ms}{0.5V}$$

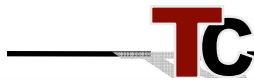
$$C = 4000\mu F$$

APPENDIX F:

Key	Decoded Output
1	0000
2	1000
3	0100
4	0010
5	1010
6	0110
7	0001
8	1001
9	0101
0	1011
*	0011
#	0111
Up	1100
Down	1110

APPENDIX G: product cost listing

Component	Quantity	Unit Price	Line Total
Keypad Encoder	1	20.00	20.00
SLA 12V Battery	1	11.89	11.89
SLA Charger	1	9.87	9.87
2V Bezel LED RED	7	0.98	6.86
2V Bezel LED GRN	2	0.98	1.96
BLU LED 900MCD	1	0.85	0.85
LED Flashing GRN	1	0.55	0.55
Rocker Switch on-off-on	1	4.10	4.10
Rocker Paddle Switch	1	2.41	2.41
Buzzer	1	1.87	1.87
Spacers	1	1.25	1.25
Cable IDC 26way	1	1.49	1.49
Skt 26way IDC	2	2.04	4.08
PLG 26way IDC	2	2.08	4.16
Aluminium	1	8.10	8.10
Enclosure GREY	2	16.43	32.86
Enclosure BLK	1	8.10	8.10
Delivery Charge - RS Components	1	11.95	11.95
Total			132.35
Amount After Uni Funding			122.35
Amount Per Person			20.39166667



menu_lcd_module.c File Reference

Contains the Menu Functions. [More...](#)

```
#include "menu_lcd_extern.h"
```

Functions

unsigned char	menu_set_min_max (unsigned char men_lvl_mm, unsigned char neg_flg) <i>Determines if the min/max/goto setting of az/el/range is within bounds.</i>
unsigned char	menu_string_select (unsigned char men_lvl, unsigned char men_dpt, unsigned char lin_flg) <i>Picks the string indexes given the current menu states.</i>
unsigned char	menu_change_state (int key_inp, unsigned char trk_flg, unsigned char men_lvl, unsigned char men_dpt) <i>Changes the menu state based on previous state and keypad input.</i>
void	LCD_string_update (unsigned char lcd_str_1, unsigned char lcd_str_2) <i>Updates the global strings to be sent to the LCD.</i>
void	LCD_update (void) <i>Updates the menu states, and displays on the LCD.</i>
void	LCD_setup (void) <i>Initialises the LCD display with our startup strings.</i>

Detailed Description

Contains the Menu Functions.

Version:

1.0

Date:

7 Jun 2006 - Completed the final product

10 June 2006 - Added Doxygen standard documentation

The menu operates through editing of state variables which contain the current menu state. These variables are updated through keypad triggers LCD is updated if and only if there is a change in the menu.

Function Documentation

void LCD_setup (void)

Initialises the LCD display with our startup strings.

Runs the initialise the LCD Module function, and then places the startup strings on the LCD: "Welcome To Team Cinder DST"

void LCD_string_update (unsigned char lcd_str_1, unsigned char lcd_str_2)

Updates the global strings to be sent to the LCD.

Parameters:

lcd_str_1 - The index of the string to be sent to the top line

lcd_str_2 - The index of the string to be sent to the bottom line

Takes the strings in rom and transfers them to the 34 bytes in ram to contain them. The strings are then updated if they have variable numbers them (such as temperature or azimuth).

void LCD_update (void)

Updates the menu states, and displays on the LCD.

When this function is called it registers that new keypad information has been recieved and then updates the menu based on that input. If the input does not change the LCD then the display is not updated.

```
unsigned char menu_change_state ( int      key_inp,
                                unsigned char trk_flg,
                                unsigned char men_lvl,
                                unsigned char men_dpt
                                )
```

Changes the menu state based on previous state and keypad input.

Parameters:

key_inp - The new keypad input (0-9,*,#,UP,DOWN and special cases)
trk_flg - The current tracking state ([0,1,2] = [Not active,Scanning,Tracking])
men_lvl - The current menu level
men_dpt - The current depth of the menu

This function takes the old menu state as well as the new keypad input and changes the menu states based on input. If the new menu is the same as the old one, a value of 0 is returned (indicating no update) otherwise a value of 1 is returned.

```
unsigned char menu_set_min_max ( unsigned char men_lvl_mm,
                                unsigned char neg_flg
                                )
```

Determines if the min/max/goto setting of az/el/range is within bounds.

Parameters:

men_lvl_mm - The current Menu Level
neg_flg - The negative flag for the current keypad value (sent as the char + or -)

This function sets the minimum/maximum/goto azimuth/elevation/range for the program by checking whether it is in the required bounds. Note: Our product requires all min angles to be less than 0, and all max angles greater than 0.

```
unsigned char menu_string_select ( unsigned char men_lvl,
                                   unsigned char men_dpt,
                                   unsigned char lin_flg
                                   )
```

Picks the string indexes given the current menu states.

Parameters:

men_lvl - The current menu level
men_dpt - The current menu depth
lin_flg - The current line

The string indexes are picked through a large switch statement (and some possibly hard to follow mathematics) given the lookup table in the repo

menu_lcd_module.c File Reference

Contains the Menu Functions. [More...](#)

```
#include "menu_lcd_extern.h"
```

Functions

unsigned char	menu_set_min_max (unsigned char men_lvl_mm, unsigned char neg_flg) <i>Determines if the min/max/goto setting of az/el/range is within bounds.</i>
unsigned char	menu_string_select (unsigned char men_lvl, unsigned char men_dpt, unsigned char lin_flg) <i>Picks the string indexes given the current menu states.</i>
unsigned char	menu_change_state (int key_inp, unsigned char trk_flg, unsigned char men_lvl, unsigned char men_dpt) <i>Changes the menu state based on previous state and keypad input.</i>
void	LCD_string_update (unsigned char lcd_str_1, unsigned char lcd_str_2) <i>Updates the global strings to be sent to the LCD</i>

Version:
1.0

Date:
7 Jun 2006 - Completed the final product
10 June 2006 - Added Doxygen standard documentation

The menu operates through editing of state variables which contain the current menu state. These variables are updated through keypad triggers LCD is updated if and only if there is a change in the menu.

Function Documentation

void LCD_setup (void)

Initialises the LCD display with our startup strings.

Runs the initialise the LCD Module function, and then places the startup strings on the LCD: "Welcome To Team Cinder DST"

void LCD_string_update (unsigned char *lcd_str_1*, unsigned char *lcd_str_2*)

Updates the global strings to be sent to the LCD.

Parameters:

lcd_str_1 - The index of the string to be sent to the top line
lcd_str_2 - The index of the string to be sent to the bottom line

Takes the strings in rom and transfers them to the 34 bytes in ram to contain them. The strings are then updated if they have variable numbers them (such as temperature or azimuth).

void LCD_update (void)

Updates the menu states, and displays on the LCD.

When this function is called it registers that new keypad information has been recieved and then updates the menu based on that input. If the input does not change the LCD then the display is not updated.

```
unsigned char menu_change_state ( int          key_inp,  
                                unsigned char trk_flg,  
                                unsigned char men_lvl,  
                                unsigned char men_dpt  
                                )
```

Changes the menu state based on previous state and keypad input.

Parameters:

key_inp - The new keypad input (0-9,*,#,UP,DOWN and special cases)
trk_flg - The current tracking state ([0,1,2] = [Not active,Scanning,Tracking])
men_lvl - The current menu level
men_dpt - The current depth of the menu

This function takes the old menu state as well as the new keypad input and changes the menu states based on input. If the new menu is the same the old one, a value of 0 is returned (indicating no update) otherwise a value of 1 is returned.

```
unsigned char menu_set_min_max ( unsigned char men_lvl_mm,  
                                unsigned char neg_flg  
                                )
```

Determines if the min/max/goto setting of az/el/range is within bounds.

Parameters:

men_lvl_mm - The current Menu Level
neg_flg - The negative flag for the current keypad value (sent as the char + or -)

This function sets the minimum/maximum/goto azimuth/elevation/range for the program by checking whether it is in the required bounds. Note: Our product requires all min angles to be less than 0, and all max angles greater than 0.

```
unsigned char menu_string_select ( unsigned char men_lvl,  
                                   unsigned char men_dpt,  
                                   unsigned char lin_flg  
                                   )
```

Picks the string indexes given the current menu states.

Parameters:

men_lvl - The current menu level
men_dpt - The current menu depth
lin_flg - The current line

The string indexes are picked through a large switch statmenet (and some possibly hard to follow mathematics) given the lookup table in the report.

3D_tracking.c File Reference

Contains the 3D Tracking and Scanning Functions. [More...](#)

Functions

void	tracking_3d (void)	<i>Performs the 3D Tracking algorithm.</i>
void	scanning_3d (void)	<i>Performs the 3D Scanning algorithm.</i>

Detailed Description

Contains the 3D Tracking and Scanning Functions.

Author:

Thomas Clement

Version:

1.0

Date:

7 Jun 2006 - Completed the final product

10 June 2006 - Added Doxygen standard documentation

The tracking and scanning algorithms change the servo angles by one increment (either one or both of azimuth and elevation) every time they are cal

Function Documentation

void scanning_3d (void)

Performs the 3D Scanning algorithm.

Moves the servos one increment of the tracking_3d_multiplier in one or both of azimuth and elevation.

void tracking_3d (void)

Performs the 3D Tracking algorithm.

Moves the servos one increment of the tracking_3d_multiplier in one or both of azimuth and elevation.



functions_shared_with_keypad.c File Reference

```
#include <pl8f452.h>
#include <stdlib.h>
#include "SERVOS.h"
#include "SERVO_EXT.h"
#include "ALL_SENSOR.h"
#include "SENSOR_EXT.h"
#include "SERIAL_STRINGS.H"
```

Functions

unsigned char	goto_function	(unsigned char two_digit_transient_buffer, unsigned char max_angle, unsigned char min_angle, unsigned char minus_flag)
unsigned char	set_azimuth	(unsigned char two_digit_transient_buffer, unsigned char set_off)
void	get_current_azimuth	(void)
void	get_current_elevation	(void)
void	get_current_range	(void)
void	enable_serial_from_keypad	(void)

Function Documentation

void enable_serial_from_keypad (void)

void **enable_serial_from_keypad**(void)

This function is called when switching from USER local mode to either factory or user remote mode.

This function does not return a value

void get_current_azimuth (void)

void **get_current_azimuth**(void)

use that function to convert the current azimuth angle into ASCII character for display stores the string of ASCII characters in a buffer and access the current position of the servos via the global variable "input_angle_azimuth"

This function does not return a value

void get_current_elevation (void)

void **get_current_elevation**(void)

use that function to convert the current elevation angle into ASCII character for display stores the string of ASCII characters in a buffer and access the current position of the servos via the global variable "input_angle_elevation"

This function does not return a value

void get_current_range (void)

void **get_current_range**(void)

use that function to convert the current distance value from the target into ASCII character for display stores the string of ASCII characters in a buffer and accesses the current position of the servos via the global variable "input_angle_elevation"

This function does not return a value

```
unsigned char goto_function ( unsigned char two_digit_transient_buffer,
                             unsigned char max_angle,
                             unsigned char min_angle,
                             unsigned char minus_flag
                             )
```

```
unsigned char set_azimuth ( unsigned char two_digit_transient_buffer,
                           unsigned char set_off
                           )
```

unsigned char **set_azimuth**(unsigned char two_digit_transient_buffer, unsigned char set_off)

This function is used to set maximum and minimum values for azimuth The function is passed the user input two_digit_transient_buffer as well as an indication as to whether or not the maximum or the minimum is being set; set_off the argument to be passed must be a 1 if the value to be set is the minimum azimuth the argument to be passed must be a 0 if the value to be set is the maximum azimuth

this function returns a wrong value if the input is invalid. This function returns the correct value if the input is valid.



initialisation.c File Reference

```
#include <pl8f452.h>
#include <stdlib.h>
#include "SERVO_EXT.h"
#include "SERVOS.h"
```

Defines

```
#define T1CON_VALUE 0b1000000
#define CCP1CON_VALUE 0b00001010
```

Functions

```
void initUART (void)
```

Define Documentation

```
#define CCP1CON_VALUE 0b00001010
```

```
#define T1CON_VALUE 0b1000000
```

Function Documentation

```
void initUART ( void )
```

```
void initUART(void)
```

This function initialises the SERIAL module, by setting the transmit line as output, the receive line as input and by writing to the corresponding registers, PIR1, PIE1, IPR1, TXSTA, RCSTA and SPBRG. This function also initialises the TIMER 1 module, which controls the servos of the device and finally Timer 2 that is used to generate a bip when an invalid input is entered through the serial interface or whenever a button on the keypad is pressed.

This function does not return a value.

keypad_module.c File Reference

```
#include "keypad_extern.h"
```

Functions

void	keypad_interrupt_setup (void) <i>Sets up the keypad interrupt.</i>
unsigned char	keypad_read (unsigned char keyp_temp) <i>The lookup table for keypad input.</i>
void	keypad_interrupt (void) <i>The keypad interrupt routine.</i>
void	update_keypad_vector (unsigned char keyp_temp, unsigned char rang_mode) <i>Updates the keypad_current_value and the keypad_current_vector with new keypad input.</i>

Function Documentation

```
void keypad_interrupt ( void )
```

The keypad interrupt routine.

Function called when the rising edge is triggered by keypad input it calls the keypad decode, and the buzzer trigger if necessary.

```
void keypad_interrupt_setup ( void )
```

Sets up the keypad interrupt.
The lookup table for keypad input.

Parameters:

keyp_temp - The input from the keypad decoder chip

Lookup table relating keypad entry to the actual value of input Errors: Returns the value 0xFF Otherwise: Returns a number from 0x00 to 0x0C 0-9 defined as 0x00-0x09 [*,#,UP,DOWN] = [0x0A,0x0B,0x0C,0x0D]

```
void update_keypad_vector ( unsigned char keyp_temp,  
                           unsigned char rang_mode  
                           )
```

Updates the keypad_current_value and the keypad_current_vector with new keypad input.

Parameters:

keyp_temp - The current input from the keypad

rang_mode - A flag representing 1 for range inputs or 0 for elevation/azimuth/factoryswitch inputs

The current entries in the keypad value/vector are updated every time this function is called. when called from the range mode, the first display character is treated as a number rather than a +/- sign otherwise it's just a 3 digit number with sign.

Mtrx 3700 File Documentation main.c

```
#include <p18f452.h>
#include <stdlib.h>
#include "SERVOS.h"
#include "SERVO_EXT.h"
#include "ALL_SENSOR.h"
#include "SENSOR_EXT.h"
#include "keypad.h"
#include "menu_lcd.h"
#include "xlcd.h"
#include "3D_tracking.h"
```

Defines

- ❖ #define **MAX_DEFAULT_ANGLE** 80
- ❖ #define **TEMP_SAMPLE_COUNTER** 199

Functions

- ❖ void **main** (void)
- ❖ void **high_interrupt** (void)
- ❖ void **low_interrupt** (void)
- ❖ void **InterruptHandlerlow** ()
- ❖ void **InterruptHandlerHigh** ()

Variables

- ❖ short int **pwm_state** = CLEAR
- ❖ int **duration_azimuth**
- ❖ int **duration2**
- ❖ int **duration_elevation**
- ❖ int **duration4**
- ❖ unsigned char **High_duration_1**
- ❖ unsigned char **Low_duration_1**
- ❖ unsigned char **High_duration_2**
- ❖ unsigned char **Low_duration_2**
- ❖ unsigned char **High_duration_3**
- ❖ unsigned char **Low_duration_3**
- ❖ unsigned char **High_duration_4**
- ❖ unsigned char **Low_duration_4**
- ❖ int **input_angle_azimuth** = CLEAR
- ❖ int **input_angle_elevation** = MAX_DEFAULT_ANGLE
- ❖ unsigned char **max_angle_azimuth** = MAX_DEFAULT_ANGLE
- ❖ unsigned char **min_angle_azimuth** = MAX_DEFAULT_ANGLE
- ❖ unsigned char **max_angle_elevation** = MAX_DEFAULT_ANGLE
- ❖ unsigned char **min_angle_elevation** = MAX_DEFAULT_ANGLE
- ❖ unsigned char **current_position_azimuth_minus_flag** = CLEAR
- ❖ unsigned char **current_position_elevation_minus_flag** = CLEAR
- ❖ unsigned char **minus_flag_azimuth** = CLEAR
- ❖ unsigned char **minus_flag_elevation** = CLEAR

- ❖ unsigned char **minus_set_azimuth_flag** = CLEAR
- ❖ unsigned char **string_number** = CLEAR
- ❖ unsigned char **rx232_counter** = CLEAR
- ❖ unsigned char **counter_tx232C** = CLEAR
- ❖ unsigned char **counter_tx232C_RAM** = CLEAR
- ❖ unsigned char **target_status_flag** = CLEAR
- ❖ unsigned char **scanning_stage** = CLEAR
- ❖ unsigned char **receive_flag** = CLEAR
- ❖ unsigned char **clear_flag** = CLEAR
- ❖ unsigned char **clear_transient_flag** = CLEAR
- ❖ unsigned char **welcome_flag** = CLEAR
- ❖ unsigned char **invalid_input_flag** = CLEAR
- ❖ unsigned char **first_character_flag** = CLEAR
- ❖ unsigned char **send_string1_flag** = CLEAR
- ❖ unsigned char **second_character_flag** = CLEAR
- ❖ unsigned char **factory_flag** = CLEAR
- ❖ unsigned char **scanning_minus_flag_azimuth** = CLEAR
- ❖ unsigned char **goto_flag** = CLEAR
- ❖ unsigned char **status_counter_string** = CLEAR
- ❖ unsigned char **send_correct_string** = CLEAR
- ❖ unsigned char **done_flag** = CLEAR
- ❖ unsigned char **string1** [BUFFER_SIZE]
- ❖ rom unsigned char **servos_not_active_string** [] = "device not active"
- ❖ rom unsigned char **scanning_enable** [2][20] = {"SCANNING DISABLED", "SCANNING ENABLED"}
- ❖ rom unsigned char **scanning_string** [] = "SCANNING FOR THE TARGET"
- ❖ rom unsigned char **tracking_string** [] = "TRACKING THE TARGET"
- ❖ rom unsigned char **welcome_string** [] = "REMOTE_MODE"
- ❖ rom unsigned char **welcome_FACTORY_string** [] = "FACTORY_MODE"
- ❖ rom unsigned char **invalid_input** [] = "Invalid Input"
- ❖ rom unsigned char **keypad_string** [] = "please use keypad"
- ❖ rom unsigned char **string_remote** [12][30] = {"0.GOTO AZIMUTH", "1.GOTO ELEVATION", "2.SET MAX AZIMUTH", "3.SET MIN AZIMUTH", "4.SET MIN RANGE", "5.SET MAX RANGE", "6.TARGET STATUS", "7.SHOW TEMPERATURE", "8.SWITCH_KEYPAD MODE", "9.SWITCH_FACTORY MODE", "s.START/STOP SCANNING", "r.REFRESH PAGE"}
- ❖ rom unsigned char **string_factory** [12][40] = {"0.CALIBRATE TEMPERATURE", "1.CALIBRATE AZIMUTH", "2.CALIBRATE ELEVATION", "3.US SAMPLE/MEASUREMENTS", "4.US SAMPLE RATE", "5.IR SAMPLE/MEASUREMENTS", "6.IR SAMPLE RATE", "7.SHOW RAW READINGS", "8.SWITCH_KEYPAD MODE", "9.REMOTE USER MODE", "S.SHOW STATISTICS", "r.REFRESH PAGE"}
- ❖ rom unsigned char **azimuth** [] = "AZIMUTH"
- ❖ rom unsigned char **elevation** [] = "ELEVATION"
- ❖ rom unsigned char **range** [] = "RANGE"
- ❖ rom unsigned char **whole_string_remote** [] = " REMOTE MODE\n\r0.GOTO AZIMUTH\n\r1.GOTO ELEVATION\n\r2.SET MAX AZIMUTH\n\r3.SET MIN AZIMUTH\n\r4.SET MIN RANGE\n\r5.SET MAX RANGE\n\r6.TARGET STATUS\n\r7.SHOW TEMPERATURE\n\r8.SWITCH_KEYPAD MODE\n\r9.SWITCH_FACTORY MODE\n\rS.START/STOP SCANNING\n\rR.REFRESH PAGE"

- ❖ rom unsigned char **whole_string_factory** [] = " FACTORY
MODE\n\r0.CALIBRATE TEMPERATURE\n\r1.CALIBRATE
AZIMUTH\n\r2.CALIBRATE ELEVATION\n\r3.US
SAMPLE/MEASUREMENTS\n\r4.US SAMPLE RATE\n\r5.IR
SAMPLE/MEASUREMENTS\n\r6.IR SAMPLE RATE\n\r7.RAW
READINGS\n\r8.SWITCH_KEYPAD\n\r9.SWITCH REMOTE
MODE\n\rS.STATS\n\rR.REFRESH PAGE"
 - ❖ int **max_range** = 2000
 - ❖ int **min_range** = 500
 - ❖ unsigned char **got_a_lock_on_target** = CLEAR
 - ❖ unsigned char **tracking_routine_flag** = CLEAR
 - ❖ unsigned char **tracking_counter** = CLEAR
 - ❖ unsigned char **next_time_toggle_negative_flag** = CLEAR
 - ❖ unsigned char **delay_counter** = CLEAR
 - ❖ unsigned char **out_of_boundary_flag** = CLEAR
 - ❖ unsigned char **overrule_flag_tracking_minus_flag** = CLEAR
 - ❖ unsigned char **first_time_counter** = CLEAR
 - ❖ unsigned char **original_input_azimuth** = CLEAR
 - ❖ unsigned char **password_minus** = CLEAR
 - ❖ unsigned char **correct_password** = CLEAR
 - ❖ unsigned char **incorrect_buzzer_counter** = CLEAR
 - ❖ unsigned char **enable_buzzer** = CLEAR
 - ❖ unsigned char **bip_lenght**
 - ❖ unsigned char **tracking_3d_flag** = 1
 - ❖ unsigned char **US_sample_counter** = 5
 - ❖ unsigned char **IR_sample_counter** = 41
 - ❖ unsigned char **correct_temperature** = CLEAR
 - ❖ unsigned char **new_input_temperature** = CLEAR
-

Define Documentation

#define MAX_DEFAULT_ANGLE 80

#define TEMP_SAMPLE_COUNTER 199

Function Documentation

void high_interrupt (void)

void InterruptHandlerHigh ()

void **InterruptHandlerHigh(void)**

This functions handles the only high priority interrupt of the program, the pwm interrupt timer1 everytime timer 1 fires, the pin_toggle function is called to generate the necessary pwms required to drive the servos
this function does not return a value.

void InterruptHandlerlow ()

void **InterruptHandlerlow(void)**

This function handles all the low priority interrupts in the program, which represents all of them except for timer 1 used for the servos.

This function does not return a value

void low_interrupt (void)

void main (void)

void **main(void)**

This is the main function of the program. Because it is entirely interrupt driven, it runs into an infinite while loop and wait for interrupts to fire and eventually alter global variables that are checked constantly inside in the loop to generate the appropriate response. The program uses timer 2 to generate an audio signal whenever an invalid input is entered through the serial connection, or whenever a key is pressed on the keypad Timer 1 is used to generate 2 independant pwm to control the servos in both azimuth and elevation. The transmit and receive interrupts are used for the serial communication, while a rise edge interrupt is used to store the input from the keypad. Timer 0 is used to fire at a definite frequency, the ultra sonic and infra red sensors.

The LCD update is not interrupt driven, at the end of every single loop, the program checks if an update is necessary or not.

In the while loop, Calculation of the distance between the target and the device as well as the temperature are carried out when the necessary flags have been set. Also it is in that main loop that the device switches between scanning and tracking mode.

This function does return a value

Variable Documentation

rom unsigned char azimuth[] = "AZIMUTH"

unsigned char bip_lenght

unsigned char clear_flag = CLEAR

unsigned char clear_transient_flag = CLEAR

unsigned char correct_password = CLEAR

unsigned char correct_temperature = CLEAR

unsigned char counter_tx232C = CLEAR

unsigned char counter_tx232C_RAM = CLEAR

unsigned char current_position_azimuth_minus_flag = CLEAR

unsigned char current_position_elevation_minus_flag = CLEAR

unsigned char delay_counter = CLEAR

unsigned char done_flag = CLEAR

int duration2

int duration4

int duration_azimuth

int duration_elevation

rom unsigned char elevation[] = "ELEVATION"

unsigned char enable_buzzer = CLEAR

unsigned char factory_flag = CLEAR

unsigned char first_character_flag = CLEAR

unsigned char first_time_counter = CLEAR

unsigned char got_a_lock_on_target = CLEAR

unsigned char goto_flag = CLEAR

unsigned char High_duration_1

```
unsigned char High_duration_2
unsigned char High_duration_3
unsigned char High_duration_4
unsigned char incorrect_buzzer_counter = CLEAR
int input_angle_azimuth = CLEAR
int input_angle_elevation = MAX_DEFAULT_ANGLE
rom unsigned char invalid_input[] = "Invalid Input"
unsigned char invalid_input_flag = CLEAR
unsigned char IR_sample_counter = 41
rom unsigned char keypad_string[] = "please use keypad"
unsigned char Low_duration_1
unsigned char Low_duration_2
unsigned char Low_duration_3
unsigned char Low_duration_4
unsigned char max_angle_azimuth = MAX_DEFAULT_ANGLE
unsigned char max_angle_elevation = MAX_DEFAULT_ANGLE
int max_range = 2000
unsigned char min_angle_azimuth = MAX_DEFAULT_ANGLE
unsigned char min_angle_elevation = MAX_DEFAULT_ANGLE
int min_range = 500
unsigned char minus_flag_azimuth = CLEAR
unsigned char minus_flag_elevation = CLEAR
unsigned char minus_set_azimuth_flag = CLEAR
unsigned char new_input_temperature = CLEAR
unsigned char next_time_toggle_negative_flag = CLEAR
unsigned char original_input_azimuth = CLEAR
```

```
unsigned char out_of_boundary_flag = CLEAR

unsigned char overule_flag_tracking_minus_flag = CLEAR

unsigned char password_minus = CLEAR

short int pwm_state = CLEAR

rom unsigned char range[] = "RANGE"

unsigned char receive_flag = CLEAR

unsigned char rx232_counter = CLEAR

unsigned char scanning_minus_flag_azimuth = CLEAR

unsigned char scanning_stage = CLEAR

rom unsigned char scanning_enable[2][20] = {"SCANNING DISABLED","SCANNING
ENABLED"}

rom unsigned char scanning_string[] = "SCANNING FOR THE TARGET"

unsigned char second_character_flag = CLEAR

unsigned char send_correct_string = CLEAR

unsigned char send_string1_flag = CLEAR

rom unsigned char servos_not_active_string[] = "device not active"

unsigned char status_counter_string = CLEAR

unsigned char string1[BUFFER_SIZE]

rom unsigned char string_factory[12][40] = {"0.CALIBRATE TEMPERATURE",
"1.CALIBRATE AZIMUTH","2.CALIBRATE ELEVATION","3.US
SAMPLE/MEASUREMENTS","4.US SAMPLE RATE","5.IR
SAMPLE/MEASUREMENTS","6.IR SAMPLE RATE","7.SHOW RAW
READINGS","8.SWITCH_KEYPAD MODE","9.REMOTE USER MODE","S.SHOW
STATISTICS","r.REFRESH PAGE"}

unsigned char string_number = CLEAR

rom unsigned char string_remote[12][30] = {"0.GOTO AZIMUTH", "1.GOTO
ELEVATION","2.SET MAX AZIMUTH","3.SET MIN AZIMUTH","4.SET MIN
RANGE","5.SET MAX RANGE","6.TARGET STATUS","7.SHOW
TEMPERATURE","8.SWITCH_KEYPAD MODE","9.SWITCH_FACTORY
MODE","s.START/STOP SCANNING","r.REFRESH PAGE"}

unsigned char target_status_flag = CLEAR
```

```
unsigned char tracking_3d_flag = 1

unsigned char tracking_counter = CLEAR

unsigned char tracking_routine_flag = CLEAR

rom unsigned char tracking_string[] = "TRACKING THE TARGET"

unsigned char US_sample_counter = 5

rom unsigned char welcome_FACTORY_string[] = "FACTORY_MODE"

unsigned char welcome_flag = CLEAR

rom unsigned char welcome_string[] = "REMOTE_MODE"

rom unsigned char whole_string_factory[] = " FACTORY MODE\n\r0.CALIBRATE
TEMPERATURE\n\r1.CALIBRATE AZIMUTH\n\r2.CALIBRATE ELEVATION\n\r3.US
SAMPLE/MEASUREMENTS\n\r4.US SAMPLE RATE\n\r5.IR
SAMPLE/MEASUREMENTS\n\r6.IR SAMPLE RATE\n\r7.RAW
READINGS\n\r8.SWITCH_KEYPAD\n\r9.SWITCH REMOTE
MODE\n\rs.STATS\n\r.REFRESH PAGE"

rom unsigned char whole_string_remote[] = " REMOTE MODE\n\r0.GOTO
AZIMUTH\n\r1.GOTO ELEVATION\n\r2.SET MAX AZIMUTH\n\r3.SET MIN
AZIMUTH\n\r4.SET MIN RANGE\n\r5.SET MAX RANGE\n\r6.TARGET
STATUS\n\r7.SHOW TEMPERATURE\n\r8.SWITCH_KEYPAD
MODE\n\r9.SWITCH_FACTORY MODE\n\rs.START/STOP SCANNING\n\r.REFRESH
PAGE"
```

Mtrx 3700 File Documentation all_01sensors.c

```
#include <p18f452.h>
#include <stdlib.h>
#include "ALL_SENSOR.h"
#include "SERVOS.h"
#include "SERVO_EXT.h"
```

Defines

- ❖ #define **AD_MAX** 1023
- ❖ #define **Var_A** 3.67
- ❖ #define **Var_B** 9400.0
- ❖ #define **Var_C** 17.0
- ❖ #define **Var_D** 2.0
- ❖ #define **Eight** 8
- ❖ #define **Five** 5
- ❖ #define **Half** 0.5
- ❖ #define **Fifty** 50
- ❖ #define **Two** 2
- ❖ #define **Min_IR** 200
- ❖ #define **Min_AVG** 300
- ❖ #define **CCP2CON_VAL** 0b00000101
- ❖ #define **T0CON_VAL** 0b00001000
- ❖ #define **T3CON_VAL** 0b10101000
- ❖ #define **ADCON1_VAL** 0b10000100
- ❖ #define **ADCON0_VAL** 0b01001001

Functions

- ❖ void **Send_pulse** (void)
- ❖ void **Calculate_distance** (void)
- ❖ void **IR_cal_distance** (void)
- ❖ void **US_setup** (void)
- ❖ void **TEMP_initialise** (void)
- ❖ void **AD_Initialise** (void)
- ❖ void **IR_Initialise** (void)
- ❖ void **get_Temp** (void)
- ❖ void **get_IR** (void)
- ❖ void **calculate_Temp** (void)
- ❖ void **Average_range** (void)
- ❖ void **clear_tracking_flags** (void)

Variables

- ❖ unsigned char **IR_flag** = CLEAR
- ❖ unsigned char **start_IR_flag** = CLEAR
- ❖ unsigned char **IR_overflows** = CLEAR
- ❖ unsigned char **send_pulse_flag** = CLEAR
- ❖ unsigned char **calculate_flag** = CLEAR
- ❖ unsigned char **no_overflows** = CLEAR

- ❖ unsigned char **start_TEMP_flag** = CLEAR
 - ❖ unsigned char **calculate_TEMP_flag** = CLEAR
 - ❖ unsigned char **IR_sample_rate** = 5
 - ❖ unsigned char **US_sample_rate** = 20
 - ❖ unsigned int **US_SR_CC**
 - ❖ unsigned int **IR_SR_CC**
 - ❖ unsigned int **IR_AD_result**
 - ❖ unsigned int **Temp_overflows** = CLEAR
 - ❖ unsigned int **temperature**
 - ❖ int **IR_ADRES**
 - ❖ int **IR_distance**
 - ❖ int **US_time**
 - ❖ int **distance**
 - ❖ int **range_value**
 - ❖ signed int **offset** = CLEAR
 - ❖ unsigned char **tracking_3d_counter**
-

Define Documentation

```
#define AD_MAX 1023

#define ADCON0_VAL 0b01001001

#define ADCON1_VAL 0b10000100

#define CCP2CON_VAL 0b00000101

#define Eight 8

#define Fifty 50

#define Five 5

#define Half 0.5

#define Min_AVG 300

#define Min_IR 200

#define T0CON_VAL 0b00001000

#define T3CON_VAL 0b10101000

#define Two 2

#define Var_A 3.67

#define Var_B 9400.0

#define Var_C 17.0

#define Var_D 2.0
```

Function Documentation

void AD_Initialise (void)

void AD_initialise(void) This function initialises the registers used for A/D CONVERSION

This function does not return a value.

void Average_range (void)

void **Average_range(void)** This function calculates the average distance between the device and the primary target by averaging out the last measurements from the ultra-sonic and infra-red sensors. These variables are accessed as global variables.

depending on the distance calculated the code changes the status of the "target_status_flag", from 1(scanning) to 2(tracking). In the mean time, this is where the transition between the tracking state and the "stay at last known location" state occurs by toggling the flag "got_a_lock_on_target".

By setting these flags(stored as global variables) high or low, this functions controls the whole tracking and scanning process of the device.

This function does not return a value.

void Calculate_distance (void)

void **Calculate_distance(void)** This function computes the distance between the object and the US sensor

This function does not return a value.

void calculate_Temp (void)

void **calculate_Temp(void)** This function is used to calculate the current temperature

This function does not return a value.

void clear_tracking_flags (void)

void **clear_tracking_flags(void)** This function clears several flags associated with the transition between tracking and scanning mode

This function does not return a value.

void get_IR (void)

void **get_IR(void)** This function is used to switch the AD channel for IR sensor

This function does not return a value.

void get_Temp (void)

void **get_Temp(void)** This function is used to switch the AD channel for temperature sensor

This function does not return a value.

void IR_cal_distance (void)

void **IR_cal_distance(void)** This function computes the distance between the object and the IR sensor

This function does not return a value.

void IR_Initialise (void)

void **IR_initialise(void)** This function initialises the registers used for IR sensor

This function does not return a value.

void Send_pulse (void)

void **Send_pulse(void)** This function prompts the US sensor to generate sound waves by setting INIT pin to high

This function does not return a value.

void TEMP_initialise (void)

void **TEMP_initialise(void)** This function initialises the registers used for Temperature sensor

This function does not return a value.

void US_setup (void)

void **US_setup(void)** This function initialises the registers used for US sensor

This function does not return a value.

Variable Documentation

unsigned char calculate_flag = CLEAR

unsigned char calculate_TEMP_flag = CLEAR

int distance

unsigned int IR_AD_result

int IR_ADRES

int IR_distance

unsigned char IR_flag = CLEAR

unsigned char IR_overflows = CLEAR

unsigned char IR_sample_rate = 5

unsigned int IR_SR_CC

unsigned char no_overflows = CLEAR

signed int offset = CLEAR

int range_value

unsigned char send_pulse_flag = CLEAR

unsigned char start_IR_flag = CLEAR

unsigned char start_TEMP_flag = CLEAR

unsigned int Temp_overflows = CLEAR

unsigned int temperature

unsigned char tracking_3d_counter

unsigned char US_sample_rate = 20

unsigned int US_SR_CC

int US_time



serial.c File Reference

```
#include <p18f452.h>
#include "SERVO_EXT.h"
#include "SERVOS.h"
#include "SENSOR_EXT.h"
#include "SERIAL_STRINGS.H"
```

Functions

void	keypad_interrupt_setup (void)
void	rx232lsr (void)
void	tx232C_RAM (unsigned char *txPtr)
void	tx232C (rom unsigned char *txPtr)
void	clear_flags (void)
void	enable_serial_receive (void)
unsigned char	get_US_counter (unsigned char US_freq)
unsigned char	get_IR_counter (unsigned char IR_freq)

Variables

unsigned char	menu_level
unsigned char	menu_depth
unsigned char	keypad_new_input_flag
unsigned char	keypad_new_entry

Function Documentation

void clear_flags (void)

void **clear_flags**(void)

This function clears several flags used in the serial MEnus

This function does not return a value

void enable_serial_receive (void)

void **enable_serial_receive**(void)

This function unmask the receive interrupt and enables the receive line for the serial connection

This function does not return a value

unsigned char get_IR_counter (unsigned char IR_freq)

unsigned char **get_IR_counter**(unsigned char IR_freq)

This function, gets the user input for the frequency at which the Infra-red sensor is firing it makes sure the new input is between 1 and 5 Hz and re a wrong value, if not

This function returns the user's input if correct

unsigned char get_US_counter (unsigned char US_freq)

unsigned char **get_US_counter**(unsigned char US_freq)

This function, gets the user input for the frequency at which the Ultra-sonic sensor is firing it makes sure the new input is between 5 and 20 Hz and return a wrong value, if not

This function returns the user's input if correct

```
void keypad_interrupt_setup ( void )
```

```
void rx232Isr ( void )
```

```
void rx232Isr(void)
```

This is the receive function for the serial communication interface. Whenever a character is sent through hyperterminal, it is saved in a buffer and analysed once the user presses the key enter. The first character sent is stored in the global variable "string_number" which corresponds to the menu chosen by the user, the next 2,3 or 4 digits are then stored in the buffer "string" and converted from ASCII to a char. A complex network of flags is used to navigate through the menu and display feedback to the user on Hyperterminal.

This function does not return a value

```
void tx232C ( rom unsigned char * txPtr )
```

```
void tx232C(rom unsigned char * txPtr)
```

This function is passed the pointer to the start of an array stored in program memory. It transmits through the serial transmit line one character after another, until it reads a NULL TERMINATOR.

This function does not return a value

```
void tx232C_RAM ( unsigned char * txPtr )
```

```
void tx232C_RAM(unsigned char * txPtr)
```

This function is passed the pointer to the start of an array stored in data memory, mainly the buffer used throughout the program "string1". It transmits through the serial transmit line one character after another, until it reads a NULL TERMINATOR.

This function does not return a value

Variable Documentation

```
unsigned char keypad_new_entry
```

```
unsigned char keypad_new_input_flag
```

```
unsigned char menu_depth
```

```
unsigned char menu_level
```

servos.c File Reference

```
#include <pl8f452.h>
#include <stdlib.h>
#include "SERVOS.h"
#include "SERVO_EXT.h"
#include "ALL_SENSOR.h"
#include "SENSOR_EXT.h"
```

Defines

#define	PWM_OFFSET1	304
#define	PWM_OFFSET2	457
#define	PWM_OFFSET3	936
#define	ONE_POINT_5ms	3750
#define	TEN_ms	0x61A8
#define	TWENTY_ms	0xC350
#define	BYTE	0x100

Functions

void	first_calculation	(void)
void	calculation	(unsigned char azi_ele)
void	pin_toggle	(void)

Define Documentation

#define BYTE 0x100

#define ONE_POINT_5ms 3750

#define PWM_OFFSET1 304

#define PWM_OFFSET2 457

#define PWM_OFFSET3 936

#define TEN_ms 0x61A8

#define TWENTY_ms 0xC350

Function Documentation

void calculation (unsigned char *azi_ele*)

void **calculation**(unsigned char *azi_ele*)

This function is called everytime the angle for azimuth and elevation are changed. The function is divided in parts. One part reserved for GOTO operation. At the end of the function, new values for the duration of the duty cycle of the two pwms are recalculated to move them to their new location.

This function does not return a value

void first_calculation (void)

void **first_calculation**(void)

This function is only called once during initialisation, as it calculates the values to write to the TIMER1 register in order to generate the appropriate pwm with regard to the input_angle_azimuth and input_angle_elevation. The method for calculating the length of the duty cycle is quite straightforward. 80 degrees = 500 ms = 1250 clock cycles 1 degrees = 1250 / 80 clock cycles For the program to be more efficient bit shifting was used instead of floating point arithmetic numbers

This function does not return a value

void pin_toggle (void)

void pin_toggle (void)

This function is called by the high priority timer 1 interrupt. Everytime that function is called, one of the two pins driving the two servos of the device is toggled. The function goes through four stages, to generate two completely independent pwms.

This function does not return a value

xlcd.c File Reference

```
#include "xlcd.h"
```

Functions

void	XLCDInit	(void)
void	XLCDCommand	(unsigned char <i>cmd</i>)
void	XLCDPut	(char <i>data</i>)
char	XLCDIsBusy	(void)
void	XLCDDelay15ms	(void)
void	XLCDDelay4ms	(void)
void	XLCD_Delay500ns	(void)
void	XLCDDelay	(void)
void	XLCDPutRomString	(rom char * <i>string</i>)
void	XLCDPutRamString	(char * <i>string</i>)

Variables

char	_vXLCDreg	= 0
------	---------------------------	-----

Function Documentation

void XLCD_Delay500ns (void)

void XLCDCommand (unsigned char <i>cmd</i>)
--

void XLCDDelay (void)

void XLCDDelay15ms (void)

void XLCDDelay4ms (void)

void XLCDInit (void)

char XLCDIsBusy (void)

void XLCDPut (char <i>data</i>)
--

void XLCDPutRamString (char * <i>string</i>)

void XLCDPutRomString (rom char * <i>string</i>)

Variable Documentation

char _vXLCDreg = 0



BLANK PAGE

